MEMORY OPTIMIZATIONS FOR DISTRIBUTED EXECUTORS IN BIG DATA CLOUDS

A Dissertation
Presented to
The Academic Faculty

By

Semih Sahin

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

May 2019

MEMORY OPTIMIZATIONS FOR DISTRIBUTED EXECUTORS IN BIG DATA CLOUDS

Approved by:

Dr. Ling Liu, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Calton Pu
School of Computer Science
Georgia Institute of Technology

Dr. Greg Eisenhauer
School of Computer Science
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. David Devecsery
School of Computer Science
Georgia Institute of Technology

Dr. Gerald Lofstead
Scalable System Software Group
Sandia National Laboratories

Date Approved: March 25, 2019

ACKNOWLEDGEMENTS

First and foremost, I owe my deepest gratitude to my supervisor, Professor Ling Liu for her encouragement, motivation, guidance and support throughout my studies. In addition to academic advising, she has shared her life lessons with me and always been available whenever I needed help. I have been extremely fortunate to have her as my doctoral advisor, and I hope to pass the lessons I learned from her on to younger generations in the future.

I would also like to express my thanks to my doctoral dissertation committee members: Professors David Devecsery, Greg Eisenhauer, Gerald Lofstead, Santosh Pande and Calton Pu. Their insightful comments and suggestions on my research have not only greatly contributed to my thesis but also helped broaden my horizons for my future research. I have also been fortunate to spend my summers as a research intern at IBM Research T.J. Watson and experience real world research and engineering challenges. I express sincere thanks to my IBM mentors and collaborators including Dr. Carlos Costa, Abdullah Kayi, Bruce D'Amora, and Yoonho Park.

I would like to thank every member of the DiSL Research Group, Databases Laboratory, and Systems Laboratory at Georgia Tech for their collaboration and companionship. It was a great pleasure to work in such a dynamic research environment. I convey special thanks to Emre Gursoy, Wenqi Cao, Lei Yu, Qi Zhang, Stacey Truex, Wenqi Wei, and Yanzhao Wu for countless research discussions and friendship. I have also been exceptionally fortunate to meet many wonderful friends in Atlanta.

I also would like to thank Abdurrahman Yasar, Erkam Uzun, Umit Catalyurek and every other members of Turkish Student Association, and people in Al-Farooq Masjid for their friendship, and the events they organized, for making myself happy, and spiritually

fulfilled.

Specially, and most importantly, I would like to thank to my father Ertuğrul, my mother Öznur and my brother Ali for always being cheerful, motivating and supportive. None of this would have been possible without their love. I am tremendously grateful for all the selflessness and the sacrifices they have made on my behalf. I am sure that, they are proud of me for this work.

I would like to make a final remark: I want to thank my advisor and all my dissertation committee members for their helpful comments and detailed suggestions and I take full responsibility of errors and inconsistencies, if any, that may remain in this document.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The amount of data generated from software and hardware sensors continues to grow exponentially as the world become more instrumented and interconnected. Our ability to analyze this huge and growing amount of data is critical. Real-time processing of big data enables us to identify frequent patterns, gain better understanding of happenings around us, and increases the accuracy of our predictions on future activities, events, and trends. Hadoop and Spark have been the dominating distributed computing platforms for big data processing and analytics on a cluster of commodity servers. Distributed executors are widely used as the computation abstractions for providing data parallelism and computation parallelism in large computing clusters. Each executor is typically a multi-threaded Java Virtual Machine (JVM) instance on Spark clusters, and Spark runtime supports memory-intensive parallel computation for iterative machine learning applications by launching multiple executors on every cluster node and enabling explicit caching of intermediate data as Resilient Distributed Datasets (RDDs).

It is well-known that JVM executors may not be effective in utilizing available memory for improving application runtime performance due to high cost of garbage collection (GC). Such situations may get worse when the dataset contains large number of small size objects, leading to frequent GC overhead. Spark addresses such problems by relying on multi-threaded executors with the support of three fundamental storage modes of RDDs: memory-only RDD, disk-only RDD and memory-disk RDD. When RDD partitions are fully cached into the available DRAM, Spark applications enjoy excellent performance for iterative big data analytics workloads as expected. However, these applications start to experience drastic performance degradation when applications have heterogeneous tasks, highly skewed datasets, or their RDD working sets can no longer fully cached in memory. In these scenarios, we identify three serious performance bottlenecks: (1) As the amount of cached data increases, the application performance suffers from high garbage collection

overhead. (2) Depending on the heterogeneity of application, or the non-uniformity in data, the distribution of tasks over executors may differ, leading to different memory utilization on executors. Such temporal imbalance of memory usage can cause out-of-memory error for those executors under memory pressure, even though other executors on the same host or in the same cluster have sufficient unused memory. (3) Depending on the task granularity, partition granularity of data to be cached may be too large as the working set size at runtime, experiencing executor thrashing and out-of-memory error, even though there are plenty of unused memory on Spark nodes in a cluster and the total physical memory of the node or the cluster is not fully utilized.

This dissertation research takes a holistic approach to tackle the above problems from three dimensions. First, we analyze JVM heap structure, components of garbage collection, and different garbage collection policies and mechanisms. Then using a variety of memory intensive benchmarks, we perform extensive evaluation of JVM configurations on application performance, under different memory sizes, heap structures and garbage collection algorithms. This comprehensive measurement and comparative analysis enable us to gain an in depth understanding of the inherent problems of JVM GC and the opportunities for introducing effective optimizations.

Second, we have engaged in a systematic study on the benefits and hidden performance bottlenecks of distributed executors and their use of RDDs in Spark runtime due to inefficient utilization of memory resources on both Spark node and Spark cluster. Through extensive measurement and analytical study, we identify several inherent problems of Spark RDDs when the partition granularity of RDDs exceeds the available working memory of the application running on Spark. To improve the performance of distributed executors for big data analytics workloads, we develop a lightweight, cooperative RDD caching framework for Spark executors. We implement the first prototype of this framework, named as DAHI. DAHI is novel in three perspectives. First, DAHI introduces a Node Manager to coordinate memory operations of JVM instances. Second, DAHI develops the d-store in-

stances that are attached to JVMs to enable coordination of data caching by the DAHI Node Manager. The combination of node manager and d-store enables DAHI to effectively reduce garbage collection overhead caused by data caching. Furthermore, the node manager coordinated RDD cache memory management provides efficient sharing of RDD caches among all JVM instances on each node of the Spark cluster. This design offers seamless capability to prevent drastic performance degradation in the situations where one or more executor experience and suffer from out-of-memory error, even though other executor(s) on the same node have plenty of unused memory. Moreover, DAHI can effectively consolidate RDD cache memory for all JVM instances on a node, and achieve high memory utilization for each executor and each cluster node by reducing or avoiding the out-of-memory errors in executors through dynamic and graceful management of RDD caching.

Finally, we propose to extend DAHI development to provide DAHI-Remote capability. Today, many data centers have reported temporal imbalance of memory utilization across nodes in a cluster, such as Google and Facebook. To address the potential problems of memory contention on some compute nodes in a cluster, DAHI-remote development is aimed to alleviate the high memory pressure of those executors that cannot find sufficient idle memory on their local nodes in a cluster by creating and providing remote memory sharing opportunities from other nodes in the cluster. The DAHI-Remote framework will enable memory coordination and caching among JVM instances across the entire cluster through a hierarchical RDD caching and memory sharing protocol. First, we enable JVM instances to have access their local native memory under local node and d-store management if available. Second, upon detection of insufficient memory for executors on the local node, DAHI-Remote creates and establish channels for remote native memory for RDD caching. For large size clusters, DAHI-Remote will create group based sharing such that each node can select and belong to one DAHI-Remote sharing group. The formation of DAHI-Remote sharing group is guided based on load balance, availability and overall performance of the applications on the cluster to ensure high throughput and low latency.

DAHI-Remote also achieves low garbage collection overhead for JVM instances, and enables efficient memory coordination among local and distributed executors across the cluster. DAHI-Remote also investigates policies for selecting remote cache node(s), selecting RDD partitioning schemes, aiming for high throughput, and fast data transfer over RDMA.

CHAPTER 1
INTRODUCTION

With the increasing demands for fast big data processing, memory is becoming one of the most critical resources to improve the performance of the various applications in Big data Clouds. Many data intensive applications, such as Key-Value stores, stream processing systems, rely on in-memory computation, memory caching, and efficient resource utilization to reach their optimal performance. Memory utilization imbalance and temporal usage variations are frequently observed in Big data clouds and production data centers.

Large memory and latency-sensitive applications enjoy linear performance surge as datasets increase in size, as long as applications can fit their working set entirely in memory. However, actual estimation and memory allocation are difficult. When these applications cannot fully fit their working sets in real memory of their executors, they suffer from large performance loss due to excessive garbage collection or page faults induced thrashing. Even when unused memory is present in other executors on the same host or a remote node in the same cluster, these applications are unable to share those unused host/remote memory.

For cloud providers, when the latency of applications fails to meet the service level agreement (SLA) or expected user experience, it could lead to notable loss in business. Google search system requires instant response time, such as a few tens of milliseconds (ms) [1], and a 500 ms delay in returning the search results could lose 20% in revenue [2]. Similarly, for Amazon.com, every 100 ms delay in page loading time could lead to 1% reduction in sales [2]. Recent study [3] [4] has shown that improving the average latency for such cloud applications is insufficient because the response time is typically dominated by the maximum latency, and the quality of service is highly related to the tail latency distribution of the cloud applications, not the mean or median latency. Although the long-

tail latency problems are known problems in networked systems [5] [6] [7], the multiple indirection at software layers in virtualized environments hurt latency sensitive cloud applications more. For example, Amazon may utilize hundreds of inter-dependent services for serving a page [8]. Some recent studies [9] [10] show that the I/O performance in Big data cloud systems has been a severe culprit to latency-sensitive, large memory workloads.

1.1 Related Research

Existing effort for handling large memory workloads can be broadly categorized into three categories.

(1) **Estimation of working set size for accurate resource allocation**. There are two inherent problems with the techniques for estimating VM/applications working set size. First, accurate memory allocation is hard as peak memory variations happen under different application types, workload inputs, data characteristics and traffic patterns. Second, applications often over-estimate their requirements or attempt to allocate for peak usage [11] [12] [13] [14] [15], resulting in severely unbalanced memory usage across VM/-containers/executors, underutilization on the host and across the cluster [11] [16] [17].

(2) **Memory-resident databases and caching**. To overcome the disk I/O bottleneck, memory resident databases [18] [19] and caching techniques [20] [21] have been proposed. Facebook caches results of frequent database queries using Memcached [20] [22]. This line of efforts by design embraces the vision [23] that main memory will be viewed as secondary storage and secondary caches to processors. These applications enjoy high throughput and low latency if they are served entirely from memory. However, actual estimation and memory allocation are difficult [24] [25] [26] [27]. Several technologies are designed to allow the NIC to send data directly to end-host applications, bypassing the OS, such as Intel DPDK [28], Open vSwitch [29], VMwares vNetwork distributed switch [30]. These technologies remove the inherent overheads due to traditional interrupt driven OS-level packet processing.

(3) **Increasing effective memory capacity**. In recent years, proposals for increasing effective memory capacity have been put forward to promote the allocation of global memory resource shared by all machines (VMs/containers/executors) to increase their effective memory capacities. These proposals promote new architectures and new hardware design for memory disaggregation [31] [32] [33] [34] [35] [36], or new programming models [37] [38]. But they lack of desired transparency at OS, network stack, or application level, hindering their practical applicability. Recent efforts represented by Accelio nbdX [39] and Infiniswap [40] exploit RDMA networks for remote memory paging with transparency. However, they adopt a flat architecture for disaggregated memory, fail to orchestrate host unused memory as the faster tier over the remote memory.

From developer perspective, domain-specific out-of-core computation techniques [41] have been implemented to juggle I/O and computation. Moreover, many large-scale web applications [22] require access to disparate parts of massive datasets for serving each user request while exhibiting little spatial locality. Amazon produces a single Web page by processing hundreds of internal requests [8], such multi-tier software architecture for application serving accumulate/aggregate the I/O latency at each tier. Given that the cost-per-GB of DRAM increases non-linearly, it is not viable in the long run to simply buy or upgrade to specialized, large-memory machines [42], which are too expensive to acquire and manage in a cost-effective manner due to the hard problem of accurate estimation of VMs working set size.

1.2 Problem Statement and Dissertation Research Scope

Hadoop and Spark have been the dominating distributed computing platforms for big data processing and analytics on a cluster of commodity servers. Distributed executors are widely used as the computation abstractions for providing data parallelism and computation parallelism in large computing clusters. Each executor is typically a multi-threaded Java Virtual Machine (JVM) instance on Spark clusters, and Spark runtime supports memory-

intensive parallel computation for iterative machine learning applications by launching multiple executors on every cluster node and enabling explicit caching of intermediate data as Resilient Distributed Datasets (RDDs). Surprisingly, most of the state-of-the-art research efforts on memory optimizations have been dedicated to big data deployment environments based on either virtual machines [43] [44] [45] [46] [47] or containers [48] [49] [50]. There are little efforts that have been devoted to improving memory resource utilization efficiency in JVM executors in a cluster computing environment, like Spark. Java Virtual Machines (JVMs) represent a class of executors that are built on garbage collected execution environment. In addition to the excellent community support and easy development capabilities, JVMs also remove the burden of explicit memory management, such as handling memory leaks and dangling pointers, from programmers. Such programming productivity gains come with the cost of relying on JVMs to managing memory and paying the automated garbage collection overhead. As a result, JVM executors are known to be less effective in utilizing available memory for improving application runtime performance due to high cost of garbage collection (GC). Such situations may get worse when the dataset contains large number of small size objects, leading to frequent GC overhead. We identify three technical challenges critical for improving JVM runtime performance:

Garbage Collection Overhead. Keeping track of pointers, and object, maintaining their lifespan and proper deallocation, are the challenges of explicit memory management. Failure of that may cause an unintended memory leaks, and performance degradation in the applications. Java Virtual Machines, however, takes that responsibility, following certain techniques to keep track of alive, and dead objects, and memory allocations, and deallocations through garbage collection. Although, this feature makes framework development easier, applications on that framework may suffer from garbage collection overhead. Since developers are no longer indicate the lifespan of an object, JVM has to execute a set of computationally expensive operations to identify live and dead objects, reclaim memory, and perform memory compaction for the live objects. Existing studies [14] [51] [52] [53] [54]

also indicate that garbage collection overhead increases as we increase the amount of data we keep in JVM heap.

Fixed Sized Executors. JVM instances (executors) are launched with fixed, predefined capacities in each node of a Spark cluster. Although this design works well with uniform tasks, there exist scenarios where some executors spill their data to disk because they do not have enough memory in their heaps, while others have idle memory in their heap. Since there is no coordination, or sharing among JVM instances on the same node in terms of memory utilization, overall application performance decreases. This understanding guides us to address the problems by designing solution approaches that can provide transparent support for elastic executors with on-demand memory coordination and native memory caching.

Memory Balancing. We observed unbalanced memory utilization across JVM executors running on both local node and in the cluster. Transient memory imbalance across JVM instances is frequently observed for skewed workloads or jobs with skewed data distributions, when JVM instances are launched with fixed, predefined heap sizes. As a result, some JVM executors are running under high memory pressure and the applications executing on these JVM instances experiencing sharp performance degradation while other executors on the same node have idle memory. But the executor that needs more memory for data caching is unable to utilize the idle memory of another executor running on the same node. Similarly, by allocating fixed size of memory to all JVM instances (e.g., Spark executors) on the same node with equal partition, one executor can have plenty of idle memory, while another may be forced to spill its RDD partitions to disk, because of insufficient memory for RDD caching. Similar problems may repeat across nodes in a cluster, such that one node has used up its available memory by its JVM executors and have to spill its data to the slow external storage such as HDD or SSD disk, while there are plenty of remote free memory elsewhere in the cluster.

In this dissertation research, we have taken a systematic approach to address such un-

balanced memory utilization at both local node and remote nodes in the same cluster. For example, by developing a layered in-memory caching structure to maximize local memory sharing and followed by remote memory sharing before resorting to slow external storage such as disk. We demonstrate that the increasing memory utilization and in-memory caching can significantly improve application runtime performance in terms of latency and throughput. Spark improves performance of iterative MapReduce applications by relying on multi-thread executors with the support of three storage modes of RDDs: memory-only RDD, disk-only RDD and memory-disk RDD [55] [56] [57]. When RDD partitions are fully cached into the available DRAM, Spark applications enjoy excellent performance for iterative big data analytics workloads as expected. However, these applications start to experience drastic performance degradation when applications have heterogeneous tasks, highly skewed datasets, or their RDD working sets can no longer fully cached in memory. In these scenarios, Spark RDD management suffer from three serious performance bottlenecks:

1. As the amount of cached data increases, the application performance suffers from high garbage collection overhead.

2. Depending on the heterogeneity of application, or working with a skewed data, the distribution of tasks over executors may differ, leading to non-uniform memory utilization on executors. Such temporal imbalance of memory usage can cause out-of-memory error for those executors under memory pressure, even though other executors on the same host or in the same cluster have sufficient unused memory.

3. Depending on the task granularity, partitions of RDDs to be cached may be too large for an executor to cache, so the entire partition is either spilled to disk, or discarded for re-computation even though there are plenty of unused memory on the local node, or other nodes in the Spark cluster.

This dissertation research takes a holistic approach to tackle the above problems. First,

we approach the design and development of our solutions by conducting an in-depth measurement study of JVM performance and its garbage collection overhead under different application workloads and with different parameter settings [58]. Second, we investigate and evaluate alternative design choices by reviewing the design space and conducting feasibility study. We propose DAHI, a node level elastic memory manager, to address the memory imbalance across JVM executors running on a single host machine. Finally, we leverage our research and development results on DAHI to explore the opportunities of utilizing remote free memory when there is insufficient local memory. We show that by intelligently combining local and remote memory sharing for RDD caching, Spark powered by DAHI and DHAI Remote can significantly improve the runtime performance of JVM executors for memory intensive applications.

Given that Spark executors are dynamically launched JVMs, and Spark is an open source platform, we demonstrate the feasibility and effectiveness of our solution approaches on top of Spark using key-value stores and machine learning workloads popular on Spark clusters. However, the design philosophy of DAHI and DAHI Remote can benefit all JVM execution environments for improving runtime performance of memory intensive applications.

1.3 Contributions and Organization

This dissertation has made three unique contributions to the state-of-the-art research and development in memory resource optimizations for distributed JVM executors in Big data clouds. We dedicate three main chapters of this dissertation to cover each of the three original contributions. In this section, we give a brief overview of the three contributions, focusing on the problems we have addressed and the design considerations and the solution approach we take to tackle the problem.

The first technical contribution is to analyze the impact of JVM configurations on the runtime performance of applications, focusing on memory utilization efficiency. Con-

cretely, we have investigated JVM heap structure, the functional components of garbage collection, and different garbage collection policies and mechanisms. We have worked with a variety of memory intensive benchmarks, performed extensive evaluation of different JVM configurations and studied their impacts on application performance, under different memory sizes, heap structures and garbage collection algorithms. Through the comprehensive measurements and comparative analysis study, we have gained some in-depth understanding of the inherent characteristics and hidden bottlenecks of JVM garbage collection (GC) and how different configuration of GC parameters may lead to different improvements on application runtime. This study has shed light on identifying the unique opportunities for the design and development of efficient memory optimizations and result in the creation of both DAHI and DAHI Remote for transparent and elastic in-memory RDD caching methodologies. Chapter 2 is dedicated to this first contribution. For example, we analyze the strengths and weaknesses of JVM execution environment on memory intensive workloads. We explore how JVM manages object allocation, and garbage collection, and the structure of heap it maintains. Our aim is to find the relation between garbage collection overhead, capacity of JVM heap, and overall application performance. We then discuss available configuration parameters, the differences between garbage collection algorithms, and indications of tuning JVM heap structure. Finally, we present extensive evaluation of these parameters on selected memory intensive benchmark applications.

The second contribution is to design and develop an elastic memory sharing system for efficient in-memory caching of RDDs in the presence of imbalanced memory utilization across executors. We started this research thrust by conducting a systematic study on the benefits and hidden performance bottlenecks of distributed executors and their use of RDDs during Spark runtime. Through extensive measurement and analytical study, we identify several inherent problems of Spark RDDs when the partition granularity of RDDs exceeds the available working memory of the application running on Spark. This understanding allows us to focus on developing more efficient in-memory RDD caching solutions, instead of

working on a much harder and more general problem: improving the JVM memory utilization for all types of Java objects. We start to design a cooperative RDD caching framework by exploiting JVM native memory as a shared memory channel. We implement the first prototype of this framework, coined as DAHI with three novel features. First, DAHI introduces a Node Manager to coordinate memory operations of JVM instances. Second, DAHI develops the d-store instances that are attached to JVMs to enable coordination of data caching by the DAHI Node Manager. Third, the combination of node manager and d-store enables DAHI to effectively reduce garbage collection overhead caused by data caching. This design offers seamless capability to provide efficient sharing of RDD caches among all JVM instances on each node of the Spark cluster. It also prevents drastic performance degradation in the situations where one or more executor experience and suffer from out-of-memory error, even though other executor(s) on the same node have plenty of unused memory. Moreover, DAHI can effectively consolidate RDD cache memory for all JVM instances on a node, and achieve high memory utilization for each executor and each cluster node by reducing or avoiding the out-of-memory errors in executors through dynamic and graceful management of RDD caching. We dedicate Chapter 3 to report the design and evaluation of DAHI. With DAHI, if there is an executor that requires more memory for data caching, under DAHI Node Manager's coordination, it can utilize idle memory that other executors are not using. DAHI also addresses overhead of garbage collection, by caching data on local native memory memory. So, it achieves high object caching, and low garbage collection overhead. We evaluate performance of DAHI using applications from machine learning, and graph processing domains from SparkBench benchmark suite.

The third technical contribution is to extend DAHI development to provide in-memory RDD caching capability by leveraging remote free memory in the cluster, referred to as DAHI-Remote. Today, many data centers have reported temporal imbalance of memory utilization across nodes in a cluster, such as Google and Facebook [17] [22]. DAHI-Remote can address the memory contention experienced on some compute nodes in a cluster when

there is idle memory on other nodes of the cluster. The DAHI-Remote framework extends the DAHI node manager to enable memory coordination and caching among JVM instances across the entire cluster through a hierarchical RDD caching and memory sharing protocol. DAHI-Remote by design has three unique features. First, we enable JVM instances to access their local native memory under local node and d-store management when there are free memory on the local node. Second, upon detection of insufficient memory for executors on the local node, DAHI-Remote creates and establishes RDMA channels to remote native memory for remote memory caching of RDD. Third, to scale to large size clusters, DAHI-Remote will create group based sharing such that each node can choose to belong to one DAHI-Remote sharing group. The formation of DAHI-Remote sharing group is guided based on a number of factors, such as load balance, availability and overall performance of the applications on the cluster. This ensures high throughput and low latency for applications and low garbage collection overhead for JVM instances, and efficient memory coordination among local and distributed executors across the cluster. DAHI-Remote also investigates policies for selecting remote cache node(s), selecting RDD partitioning schemes, aiming for high throughput, and fast data transfer over RDMA. We dedicate Chapter 4 to report our design and development of DAHI-Remote. For example, we discuss the underlying technology behind RDD transport layer, including TCP/IP, and RDMA based solutions. We also explore common policies required in such system, including remote node selection, streaming write/read operations to reduce latency, and ownership updates to minimize data movement among nodes over the network. With DAHI-Remote, RDD partitions are cached in local native memory first, if there is not enough memory in local node, then partitions are cached in remote node, if there is no memory available in the entire cluster, then data is either spilled to disk, or discarded for re-computation.

We conduct experimental evaluation on Linear Regression and Connected Components applications from SparkBench benchmark suite and our results validate the effectiveness of DAHI-Remote.

1.4 Thesis Statement

This dissertation research advances the field of memory utilization efficiency and brings significant performance gains for big data and machine learning workloads in Big Data Clouds.

The thesis of this dissertation is that the performance of JVM based executors which relies on data caching for iterative applications, can be improved by enabling them to have elastic and coordinated memory utilizations, both in node level and cluster level. While elasticity is achieved with explicit caching in native/off-heap memory, which also reduces the garbage collection overhead, memory coordination allows executors to utilize total available memory in the cluster more effectively to increase caching performance, by enabling executors to cache their data in idle memory on other nodes in the cluster over RDMA, addressing problems caused by unbalanced memory utilization among executors, and nodes.

CHAPTER 2

JVM CONFIGURATION MANAGEMENT AND ITS PERFORMANCE IMPACT FOR BIG DATA APPLICATIONS

Big data applications are typically programmed using garbage collected languages, such as Java, in order to take advantage of garbage collected memory management, instead of explicit and manual management of application memory, e.g., dangling pointers, memory leaks, dead objects. However, application performance in Java like garbage collected languages is known to be highly correlated with the heap size and performance of language runtime such as Java Virtual Machine (JVM). Although different heap resizing techniques and garbage collection algorithms are proposed, most of existing solutions require modification to JVM, guest OS kernel, host OS kernel or hypervisor. In this chapter, we evaluate and analyze the effects of tuning JVM heap structure and garbage collection parameters on application performance, without requiring any modification to JVM, guest OS, host OS and hypervisor. Our extensive measurement study shows a number of interesting observations: (i) Increasing heap size may not increase application performance for all cases and at all times; (ii) Heap space error may not necessarily indicate that heap is full; (iii) Heap space errors can be resolved by tuning heap structure parameters without enlarging heap; and (iv) JVM of small heap sizes may achieve the same application performance by tuning JVM heap structure and GC parameters without any modification to JVM, VM and OS kernel. We conjecture that these results can help software developers of big data applications to achieve high performance big data computing by better management and configuration of their JVM runtime.

2.1 Introduction

Big data computing platforms today are represented by Hadoop HDFS/MapReduce [59], Spark [55] and Storm [60]. These platforms are delivering software as a service, written in garbage-collected languages, e.g., Scala, Java, C#, ML, JRuby, Python. The execution environment of garbage collected languages typically involves allocation of portion of memory to each application, and managing the allocated memory and garbage collection (GC) on behalf of the applications at runtime. This removes the burden of explicit memory management from the developers, e.g., handling memory leaks, removing dangling reference pointers and dead objects. Thus, garbage collected languages, such as Java, become increasingly popular and Java Virtual Machines (JVMs) are recognized as a leading execution environment for many big data software platforms today.

However, running applications on JVMs presents another layer of abstract execution environment on top of hardware virtualization (i.e., virtual machines and hypervisor) over a physical hosting server platform. Thus, the dynamic sharing of physical CPU and memory among multiple JVMs, combined with the unpredictable memory demands from applications, creates some open challenges for JVM configuration management. Most frequently asked questions include: (1) How can applications balance the sizes of their JVM heaps dynamically? (2) What type of heap structure can minimize heap space errors? (3) Which set of JVM heap parameters can we use to control the garbage collection overheads and improve application runtime performance? (4) Can we speed up the progress of application by enlarging the JVM heap size?

Researchers have attempted to address some of these questions. [61] shows that one can obtain the same performance by either explicit memory management or using garbage collection, provided that the garbage collected heap is sized appropriately with respect to the application. However, keeping the heap an appropriate size and balancing the heap sizes dynamically in a multi-application runtime environment are extremely difficult. Small heap

will trigger frequent garbage collection, which results in performance degradation. Also collecting heap too frequently increases garbage collection (GC) cost. Moreover, small heap may lead to more frequent heap space errors, causing the application with dynamic memory demand to fail. On the other hand, heap can only be set to larger size when memory is plenty. Also too large a heap can induce paging, and swapping traffic between memory and disk is several orders of magnitude more expensive, significantly degrades the performance of the system. Several resource management technologies, such as memory overcommitment, heap resizing, and virtual machine memory ballooning, are proposed to address the problem of sudden changes in memory demands of applications in a consolidated environment. However, most of existing methods require modifications to JVM, guest virtual machine kernel or host OS kernel and hypervisor.

In this chapter, we evaluate and analyze the effects of tuning JVM heap structure parameters and garbage collection parameters on application performance, without requiring any JVM, guest OS, host OS or hypervisor modification. Through our extensive measurement study, we show a number of valuable observations that help answering the set of questions listed above. First, we show that there is a direct correlation between application performance and its heap size. However, after heap size reaches a certain value, increasing heap size no longer improves the application performance in terms of the amount of actual work done. Second, we show that heap space error does not necessarily correspond to the conclusion that heap is full and all objects residing in heap are alive. Also, heap space error can be resolved by tuning JVM heap structure parameters. Third, dynamic configuring of JVM heap structure parameters and GC parameters may help minimizing GC overheads and improving application runtime performance. Our experiments also show that smaller heap sizes can achieve the same level of application performance as some of the larger heap sizes. Finally, we show that most of our measurement results are not specific to any garbage collection algorithm or any specific garbage collector implementation. We conjecture that these measurement results can help software developers of big data applications to better

manage and configure their JVM runtime, achieving high performance big data computing at ease.

2.2 Related Work

Memory management research related to JVM performance can be broadly categorized into three categories: Memory overcommitment techniques, heap resizing algorithms and memory bloat management.

Memory overcommitment. Memory overcommitment is used at hardware virtualization layer by enabling virtual machines (VMs) to inflate and deflate memory using Balloon driver [62]. However, sharing policies are insufficient to determine when to reallocate memory and how much memory is needed. [43] proposes Memory Balancer (MEB), which dynamically monitors memory usage of each VM, and reallocates memory by predicting memory need of each VM. [63] proposed an application level ballooning technique, which enables resizing JVM heap dynamically, by modifying heap structure. However, they do not provide any sharing or resizing policy. [64] described a JVM ballooning by allocating Java objects. Balloon objects are then inflated/deflated depending on the resizing decision.

Heap Resizing. Independent but complimentary to the development of Java ballooning mechanism, several research efforts have been devoted to heap resizing policies. [53] evaluated JVM metrics as application performance indicator, and proposes a memory sharing policy by considering applications memory demands and available physical memory. [51] proposed CRAMM to track memory demands of applications at runtime, and predict appropriate heap sizes, aiming at garbage collection without paging or minimizing paging while maximizing application throughput. [52] presented a resource aware garbage collection technique with different heap resizing policies. [14] proposed Ginkgo, a policy framework to correlate application performance and its memory demand by using runtime monitoring. However, few existing efforts have conducted a systematic study on JVM configuration management with respect to heap structure parameters and GC parameters, such as show-

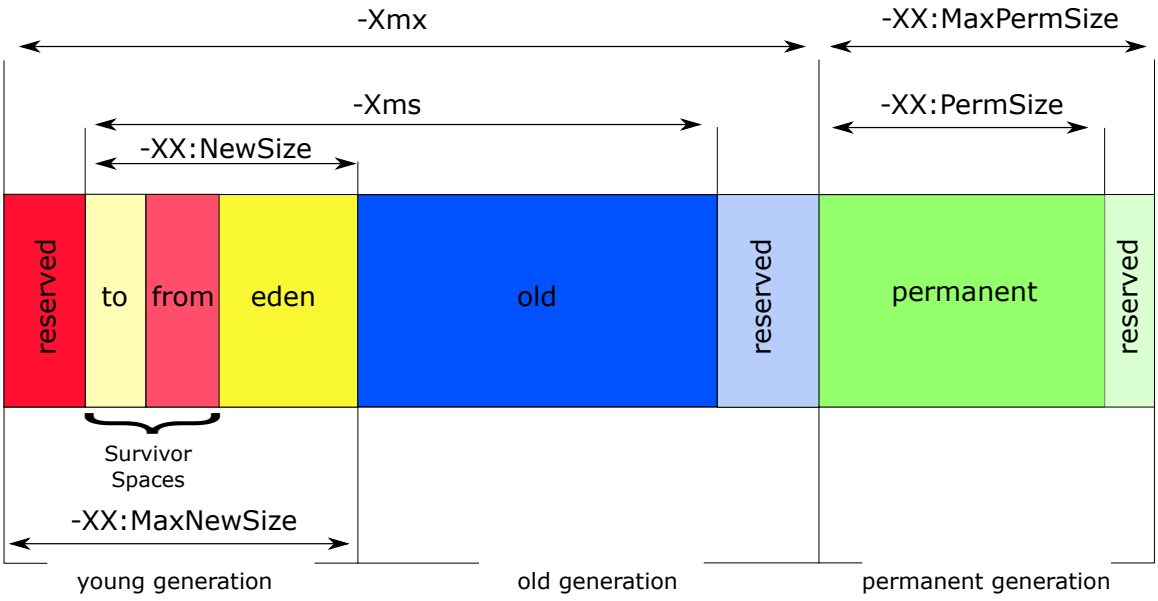


Figure 2.1: JVM Structure

ing the effect of changing the size ratios between young generation and old generation, and understanding the root cause of heap space errors.

Memory Bloat Management. To improve the runtime efficiency of JVM heap, researchers have identified different classes of memory bloats caused by data type design [65, 66] and proposed new design for JVM data types that can minimize or remove undesirable memory bloats.

2.3 Overview and Observations

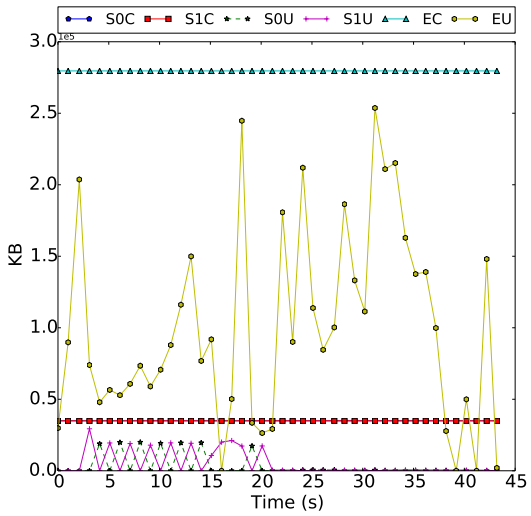
2.3.1 JVM Heap Structure

In JVM, memory management is done by partitioning the memory region, called JVM heap, into groups, where each group consists of one or more regions and is called a generation. Objects in one generation are of a similar age. Figure 2.1 shows JVM memory structure. Most commonly, there are two major types of generations: young and old. In **young generation**, recently created objects are stored. Typically, young generation is further partitioned into 3 major spaces (regions): *eden* space and two survivor spaces: *to* and *from*. New objects are first allocated in *eden* space. The allocation is done contiguously,

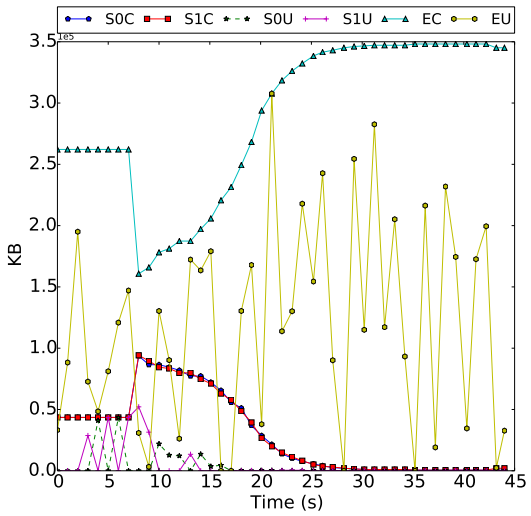
enabling fast placement. Minor GC is triggered on allocation failure, whereas Full GC is triggered when meeting a threshold, i.e., some percentage of *old* generation has been filled. Upon a minor GC, objects that are not matured enough to move to old generation, but survived at least one garbage collection are stored in *from* survivor space. The *To* survivor space is unused.

In **old generation**, objects that are survived a certain number of garbage collection are stored. By default the old generation is twice as big as young generation. Keeping young generation small makes it collected quickly but more frequently. Old generation grows more slowly but when reaches to certain threshold value, a full (heap) garbage collection (GC) will be triggered. The reserved region in both young and old generations and the permanent generation are to allow JVM to expand its heap. The default setting of old/young capacity ratio is fixed and 2. However, capacities of spaces in young generation may vary from one garbage collector to another. For derby workload with heap size of 1024 MB, Figure 2.2 shows that while the default capacity ratio of eden and survivor spaces is fixed for Serial GC or Concurrent Mark Sweep (CMS) Garbage Collector, but the capacities are dynamically changing at runtime for Parallel GC Collector or Parallel Old Garbage Collector.

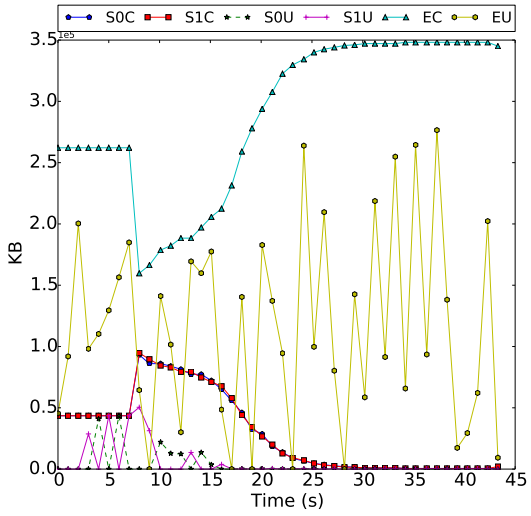
Figure2.3, produced using Visual VM [67], shows the utilization of eden and survivor spaces in young generation and the utilization of old generation for Dacapo h2 workload with heap sizes of 1024 MB and 384 MB. We make three observations: (1) For heap size of 1024MB, in young generation, Eden space is 94% full (257.670/273.062=0.94) and Survivor 1 is 74% full (25.237/34.125=0.739) while Survivor 0 is empty. The utilization of old generation is 27% (183,830/682.688=0.269). (2) For heap size of 384 MB, in young generation, Eden space is 85% full, while the utilization of old generation is over 81% (208.300/256.000=0.813). (3) In both cases, a number of minor GC and full GC are performed and the time spent on minor GC and full GC are also measured: 92 minor GCs are performed for heap size of 384 MB, compared to 29 minor collections for heap size of



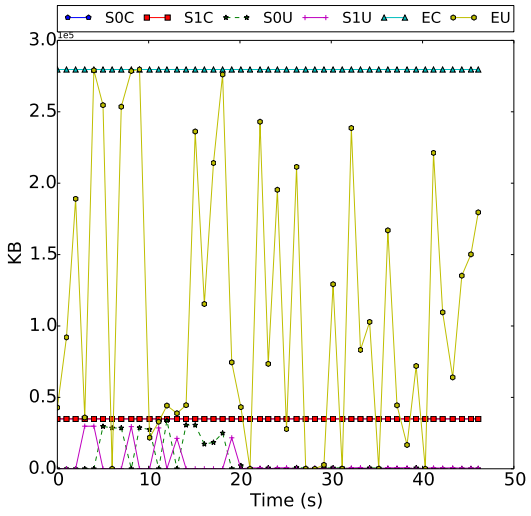
(a) SerialGC



(b) ParallelGC

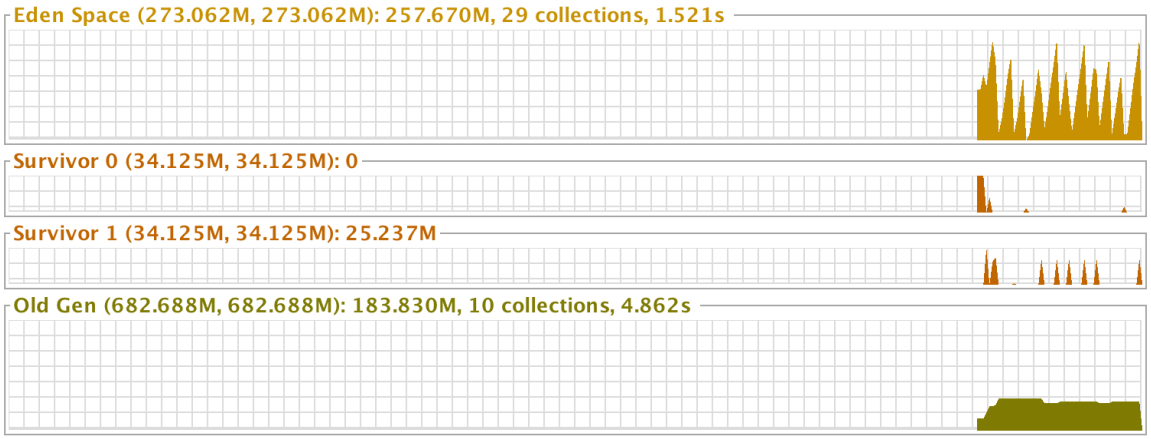


(c) ParallelOldGC

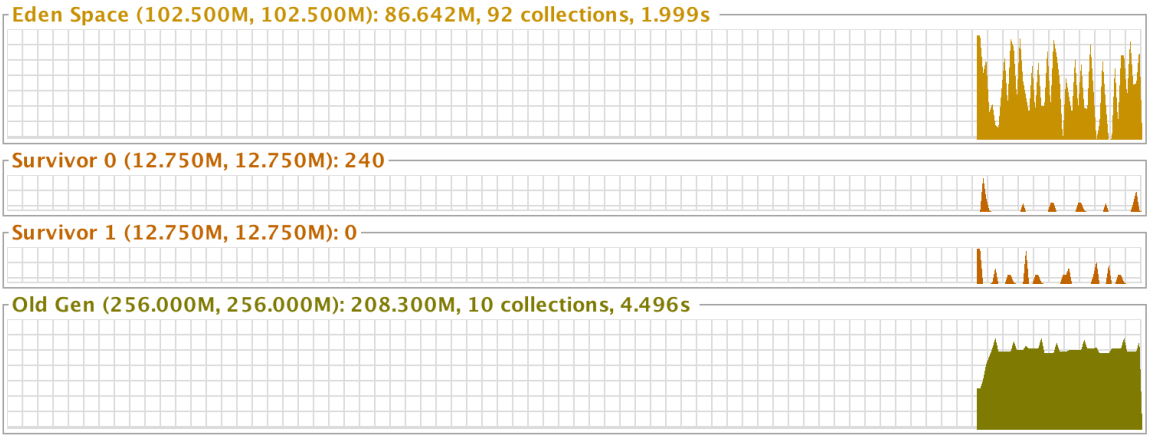


(d) ConcMarkSweepGC

Figure 2.2: Capacities and Utilizations of Young Generation (S0C/SIC: capacity of survivor space 0 / survivor space 1, EC/EU: eden space capacity/utilization, S0U/S1U: utilization of survivor space 0 / survivor space 1).



(a) Heap Utilization with Heap Size=1024MB



(b) Heap Utilization with Heap Size=384MB

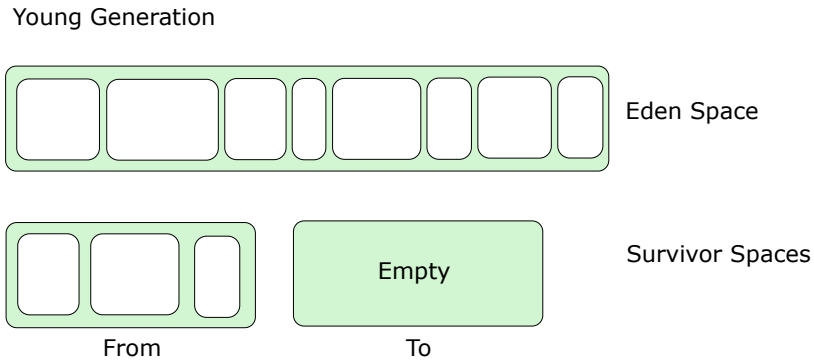
Figure 2.3: Heap Utilization of Dacapo h2 Workload

1024 MB, though both have 10 full GCs.

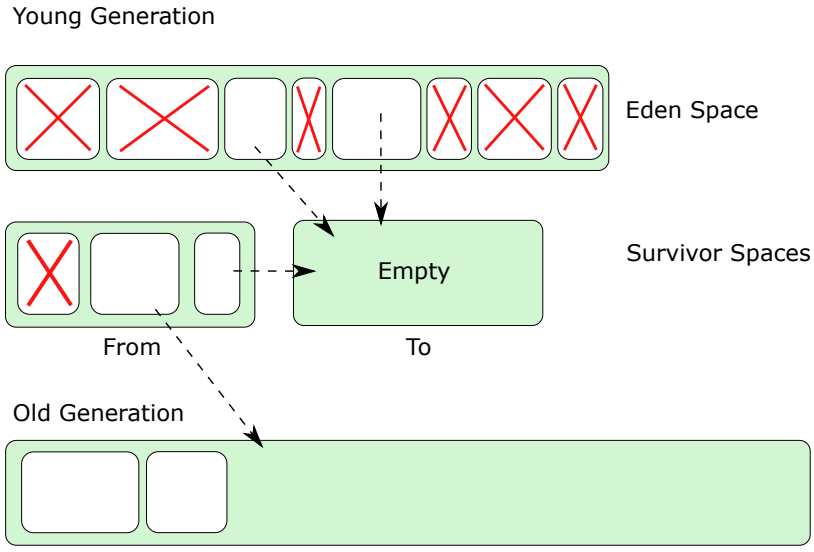
2.3.2 Garbage Collection (Minor GC vs Full GC)

Garbage collection is the process of identifying dead objects, which are no longer referenced by application and reclaim their heap space. Garbage collector is also responsible for object allocation and it can be performed on generations separately or on the entire heap.

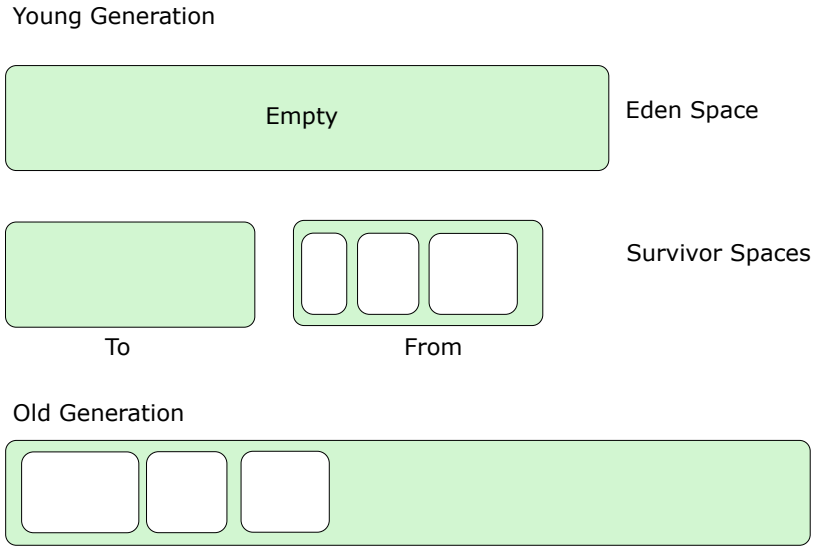
Young generation Minor GC is triggered when new object allocation fails because there is insufficient space in young generation. During Minor GC, dead objects are identi-



(a) Before GC

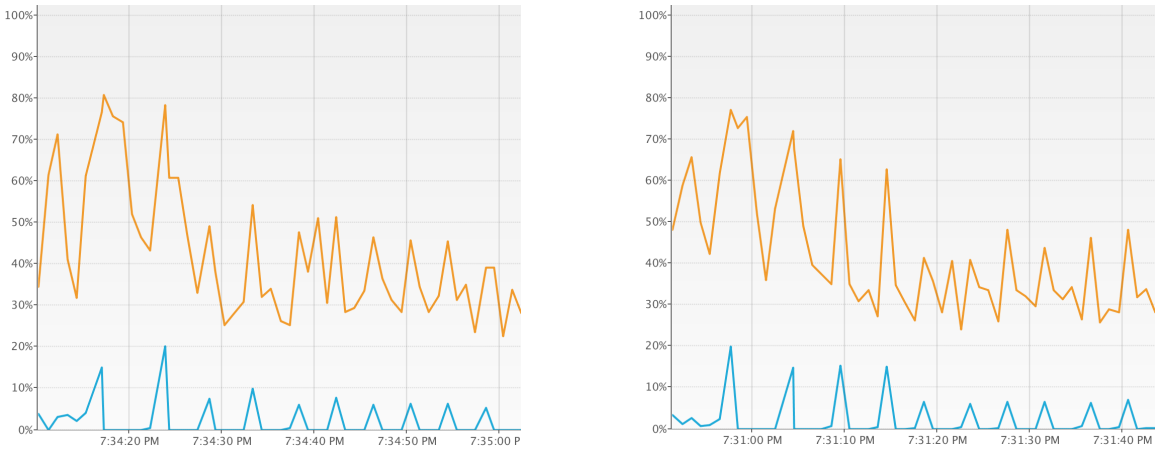


(b) During GC



(c) After GC

Figure 2.4: Garbage Collection Process (Minor GC)



(a) Heap Size = 1024MB

(b) Heap Size = 384MB

Figure 2.5: CPU Utilization of h2 Workload

fied, and live objects in eden space, which are not mature enough to move to *old generation*, are moved to the survivor *to* space, which was previously unused. Live objects in the survivor *from* space are moved either to *to* space or *old generation* based on their ages. After minor GC, survivor *from* space becomes the survivor *to* space, and vice versa. In our experiments, Serial Garbage Collector is used during Minor GC by default. Figure 2.4 illustrates how it works when Minor GC is triggered. Allocation triggers minor GC in young generation when there is insufficient space in Eden to place a new object. Threshold triggers full GC in old generation when heap usage in old generation reaches a specified threshold, **Full GC** is performed on entire heap, the most costly collection compared to Minor GC. It is also more complex because there is no helper space, like survivor space for Minor GC, to enable compaction. A popular policy used in Full GC is called *mark-sweep-compact*, in which dead objects are identified first and live objects are moved to the head of old generation. Once they are placed contiguously, the rest of the heap is reclaimed. Both Minor GC and Full GC suspend applications from executing and take full control of CPU when switched on. Figure 2.5 shows CPU usage of application (blue curve) and garbage collection (yellow curve) running concurrently for h2 workload with heap sizes of 1024 MB and 384 MB.

2.3.3 Garbage Collection Overhead

Full GC and minor GC both stop the execution of JVM applications when performing garbage collection. There are a number of factors contribute to the GC overhead. For simplicity, we focus our discussion on young generation collections. As stated earlier, minor GC is performed when an allocation fails, namely, when there is insufficient space in the eden space to meet the allocation request. Depending on the size of the new object, minor GC can be triggered with varying sizes to be collected from the eden space. Since both object allocations and GC preserve compaction of the heap regions, GC is performed only on the utilized portion of the Eden space. Hence, GC overhead depends on the frequency of allocation failure and the eden space utilization.

In general, GC is more centered on identification of alive objects, instead of dead ones. For the sake of compaction, it involves copying alive objects from eden to survivor, or from young generation to old generation. As a result, copying cost emerges as the third factor in GC overhead.

These factors can be validated by GC report, which can be obtained by adding **-XX:+PrintGCDetails** parameter to the run command. In Table 2.1, we present some GC details performed on young generation for compiler.compiler benchmark application with heap size of 512MB. It shows that 1 time GC cost is proportional to utilized eden space and size of the alive data.

Parameter Tuning for Heap Structure and GC. JVM heap structure and garbage collection can be configured by tuning the following JVM parameters:

-Xmx, -Xms parameter specifies the maximum and minimum heap size that JVM can manage respectively.

-XX:NewRatio=n sets ratio of old/young generation sizes to n. Default NewRatio value is 2.

-XX:+UseSerialGC enables the serial garbage collector. **-XX:+UseParallelGC** activates parallel garbage collector for the young generation.

Table 2.1: Garbage Collection Details

Eden Util. (kb)	Alive Size (kb)	Garbage Size (kb)	GC Cost (msec)
139776	9877	129899	68,1877
139776	10291	129485	73,4889
139776	10691	129085	71,1739
139776	10798	128978	84,3332
157248	10163	147085	124,8909
157248	12556	144692	148,5655
157248	17472	139776	221,4381

-XX:+UseParallelOldGC enables the parallel garbage collector for both of the young and old generations.

-XX:+UseMarkSweepGC activates concurrent mark sweep garbage collector for the old generation.

2.4 Experimental Method

2.4.1 Experimental Environment

In all of our experiments, we disable *UseGCOverheadLimit* parameter since we want to focus on application performance in any case unless it results in heap space error. We use Oracle's HotSpot Java Runtime Environment with version 1.8.0_65. In addition, we use jstat and VisualVM to monitor heap usage and CPU usage of our benchmark applications (see next subsection). Measurement period is set to 1 second. Unless otherwise is stated, in our experiments we use Serial Garbage Collector with default newRatio value 2. The

hardware platform is a Mac OSX Yosemite 10.10.5, with two physical cores of 2.9 GHz Intel i5 and 1867 MHz DDR3 memory of 8 GB.

2.4.2 Test Programs

We select 4 benchmark applications from DaCapo [68] and SPECjvm2008 [69] benchmark suites without any modification. Here are the chosen benchmarks:

h2: As part of DaCapo suit, h2 executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application. Unless otherwise is stated, h2 benchmark is run 10 times.

derby: This benchmark uses an open-source database written in pure Java. It is synthesized with business logic to stress the BigDecimal library. The focus of this benchmark is on BigDecimal computations and database logic.

serial: This benchmark serializes and deserializes primitives and objects, using data from the JBoss benchmark. The benchmark has a producer-consumer scenario where serialized objects are sent via sockets and deserialized by a consumer on the same system.

compiler.compiler: This benchmark uses OpenJDK front end compiler to compile a set of files with *.java* extension. The code compiled is javac itself and the sunflow subbenchmark from SPECjvm2008.

2.4.3 Collectors and Configuration Parameters

In all experiments, SerialGC is used as the baseline collector. It uses a single thread with copying and compaction preserving mechanisms for young generation. It stops all applications when performing minor GC. We compare SerialGC with ParallelGC, ParallelOldGC and the Concurrent Mark-Sweep (CMS) Collector. ParallelGC is similar to SerialGC but performing minor GC using multiple threads instead of single thread. ParallelOldGC uses copying and compaction preserving mechanisms for GC in both young generation and old generation and is multi-threaded. The CMS Collector marks the reachable objects and

then sweeps across the allocation region to reclaim the unmarked objects (spaces). It is non-copying and non-compacting in that it does not move allocated objects and neither compact them. In contrast, copying collectors proceed by copying reachable objects to an empty copy space. Also upon GC, it suspends application only at the beginning and end of the collection marking and sweeping phases. CMS GC are concurrent with the application executions (no stop-the-world) and it uses multiple threads.

2.4.4 Metrics

The first performance metric is the ratio of old generation over the new generations. By default of 2, it states that the old generation is doubled the size of the young generation. In most of our experiments, we vary this ratio from 1 to 21. Another important metric is the GC overhead, consisting of three components: allocation failure frequency, utilization of corresponding generation, and size of alive objects that will create cost of copying. Let n denote number of GC as a frequency indicator, α_i denote the size of garbage data and β_i denote the size of alive objects in the i^{th} GC event. Let c_1, and c_2 denote machine specific constants for scanning and copying heap unit respectively. Then we can calculate GC overhead as follows:

$$GC\_Overhead = \sum^{n} c_1(\alpha_i + \beta_i) + c_2(\beta_i)$$

Other metrics in our measurement study include heap size, heap utilization, execution time, CPU utilization.

2.5 Experiments and Analysis

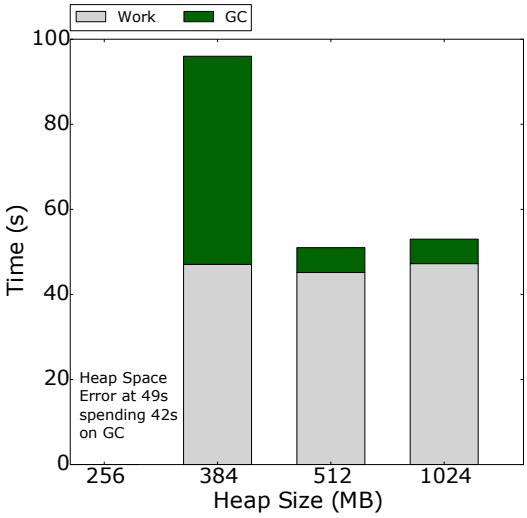
In this section, we evaluate the effect of JVM heap size, structure and GC algorithm on memory intensive benchmark applications by tuning above JVM parameters.

Table 2.2: Heap Utilization at the time of Heap Space Error

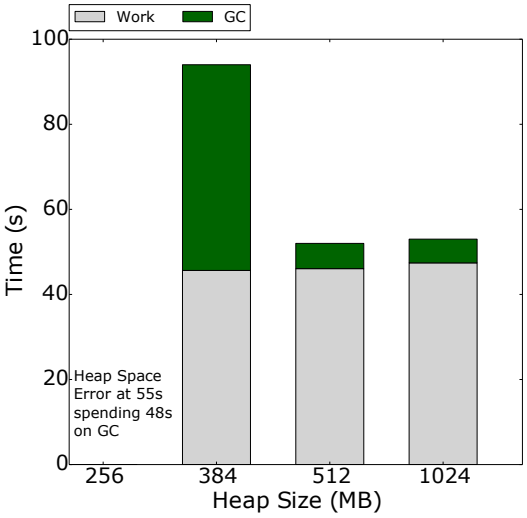
Benchmark	GC	Eden Cap.	Eden Used	Surv. 0 Cap.	Surv. 0 Used	Surv. 1 Cap.	Surv. 1 Used	Old Cap.	Old Used
h2	ParallelGC	44032.0	44032.0	16384.0	0	43520.0	0	131072.0	131072.0
h2	ParallelOldGC	44032.0	44032.0	16384.0	0	43520.0	0	131072.0	131072.0
derby	ParallelOldGC	88064.0	88064.0	19968.0	0	20992.0	0	131072.0	131040.6
compiler.compiler	ParallelGC	22528.0	22528.0	21504.0	0	21504.0	0	65536.0	65493.5
compiler.compiler	ParallelOldGC	22528.0	22527.3	21504.0	0	21504.0	0	65536.0	65533.6

Table 2.3: GC Overhead with Varying NewRatio Values

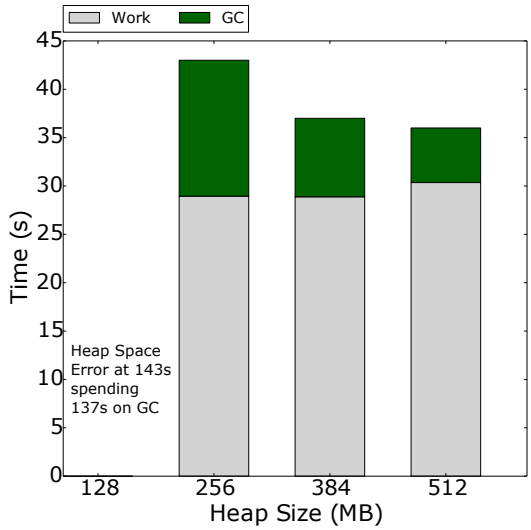
newRatio	# of YoungGC	YoungGC Cost	# of FullGC	FullGC Cost	Total GC Cost
1	18	0.383	824	121.720	122.103
2	36	0.510	752	107.006	107.516
3	63	0.617	717	98.442	99.059
5	1192	2.315	12	1.499	3.815
8	1788	2.946	6	0.704	3.650
13	2788	4.365	4	0.357	4.722
21	4344	5.926	4	0.354	6.280



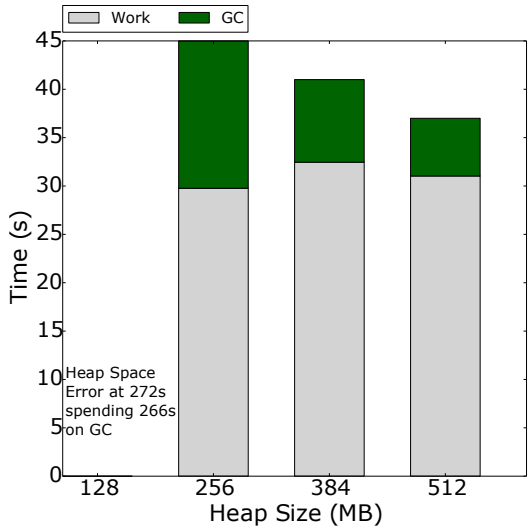
(a) h2-ParallelGC



(b) h2-ParallelOldGC



(c) compiler.compiler-ParallelGC



(d) compiler.compiler-ParallelOldGC

Figure 2.6: Heap Space Error Elimination by Increasing Heap Size

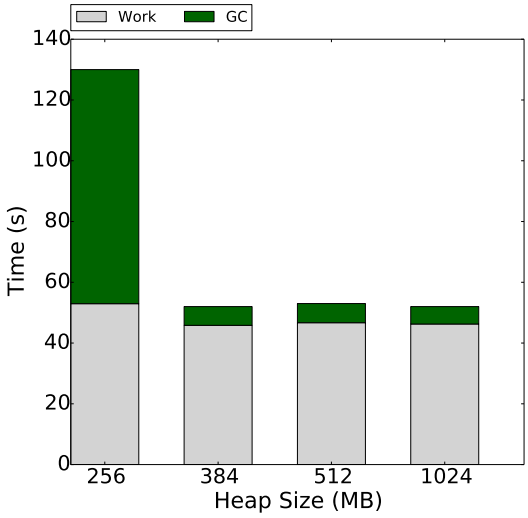
2.5.1 Heap Utilization and Heap Space Error

In this set of experiments, we analyze the scenarios that causes heap space error. One may think that the reason behind heap space error is failing object allocation because the entire heap is filled and all the objects in it are alive and no additional space to reclaim. However, our experiments show that depending on the GC algorithm, the size of the survivor and eden spaces may change at runtime. Since object is allocated only in eden space or old generation, allocation may fail when eden space or old generation is filled, resulting a heap space error. However, there could be some unused survivor spaces that are not small. Table 2.2 illustrates this observation, where we set newRatio value to 1, and heap size to 256MB for h2 and derby benchmarks, and 128 MB for compiler.compiler.

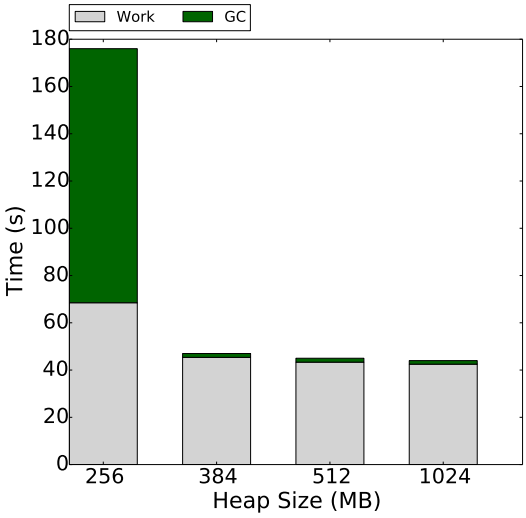
2.5.2 Effect of Heap Size and GC Overhead

The most straightforward approach to eliminate Heap Space Error is to increase JVM heap size. Figure 2.6(a)(b) show that (i) JVM crashed for heap size of 256MB, and (ii) increasing heap size from 256MB to 384MB resolves Heap Space Error, for h2 workloads with Parallel and ParallelOld GC respectively. Figure2.6(c)(d) show that JVM crashed for heap size of 128MB, and (ii) increasing heap size from 128MB to 256MB resolves Heap Space Error for compiler.compiler benchmark. In all these figures, the smallest heap size is too small to run the respective benchmark, which causes heap size error and crashed.

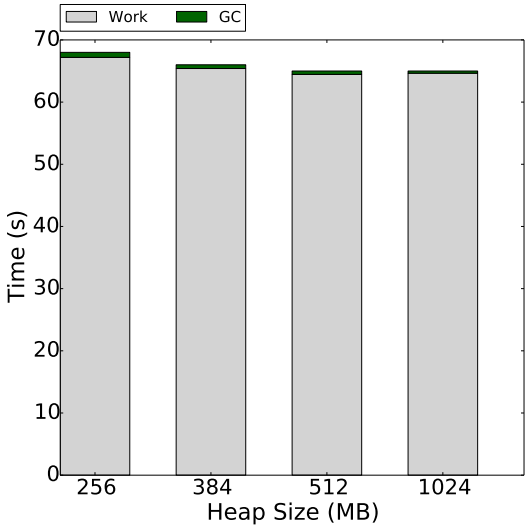
Next we examine the effects of heap size. Increasing heap size leads to less frequent allocation failure and improves application performance by reducing GC frequency and thus overall GC overhead. Figure 2.7a and Figure 2.7b show that by increasing heap size from 256MB to 384MB, it reduces GC overhead drastically for derby and h2 workloads. We also see similar result in Figure 2.7d for compiler.compiler benchmark when we increase heap size from 128MB to 256MB. We also observe that after some point, increasing heap size no long reduces GC overhead, nor improve application performance. In summary, we showed that increasing heap size may eliminate Heap Space Error, and improve application



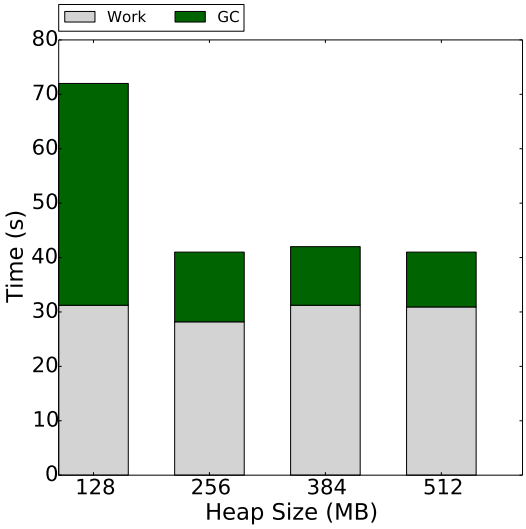
(a) h2



(b) derby

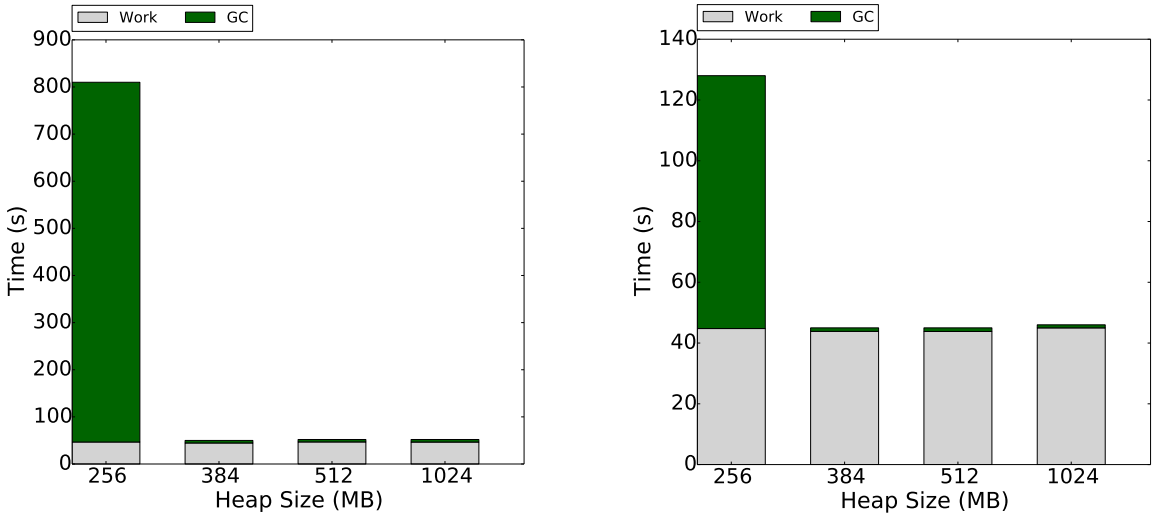


(c) serial



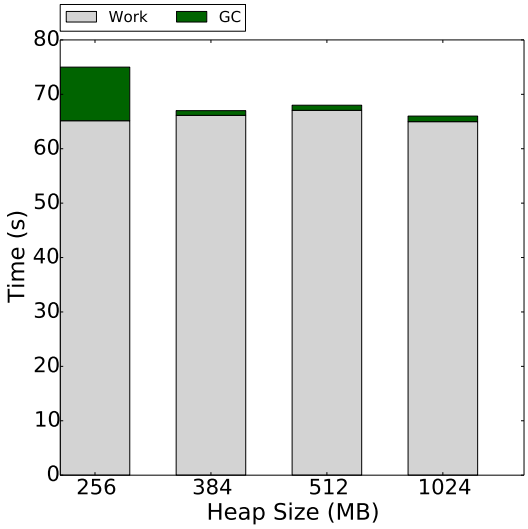
(d) compiler.compiler

Figure 2.7: Heap Size vs Running Time (SerialGC)

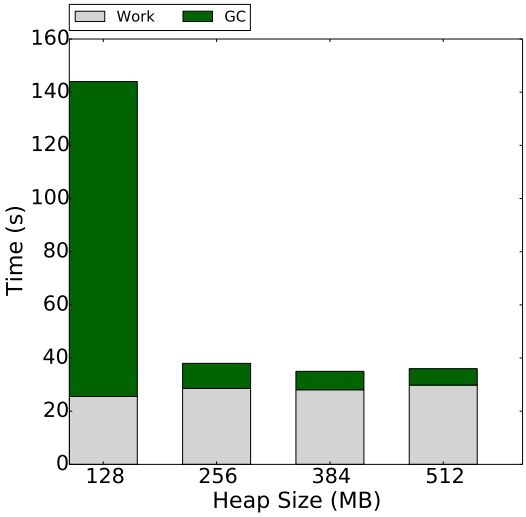


(a) h2

(b) derby



(c) serial



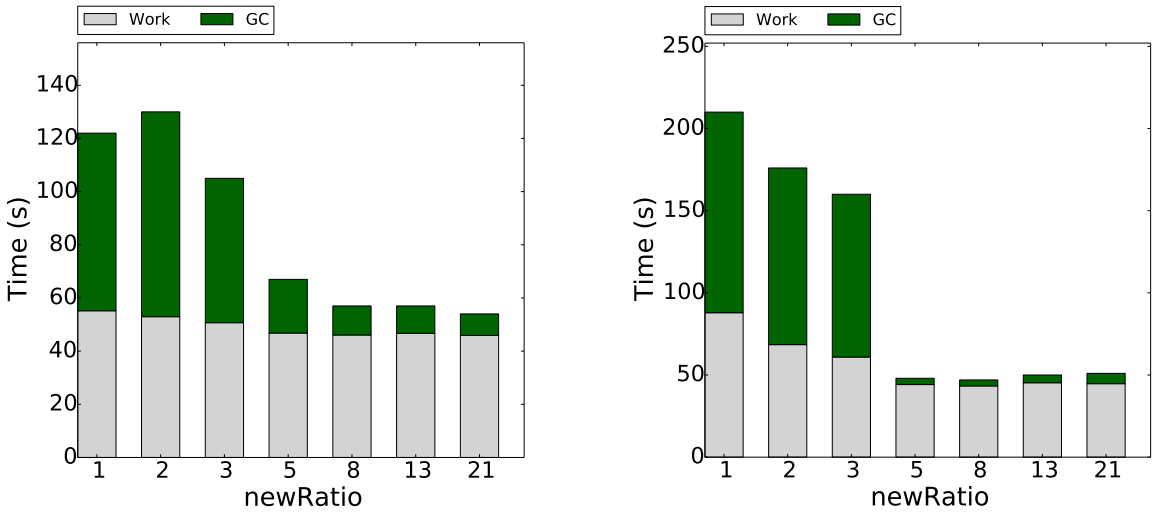
(d) compiler.compiler

Figure 2.8: Heap Size vs Running Time (ParallelGC)

performance by reducing GC overhead. However, increasing heap size may not be possible in a consolidated environment where memory is a primary bottleneck. This motivates us to examine other tuning options that explore more efficient heap utilization.

2.5.3 Tuning newRatio Parameter

Another cause for Heap Space Error is due to poor heap utilization in JVM. This is the case when Heap Space Error is experienced but we can still observe some large but unused survivor spaces as shown in Figure 2.3. Our goal is to reduce the size of unused survivor spaces, so that more portion of the heap is used for allocation to objects. One way to achieve this goal is to increase the parameter newRatio value, which will increase the size of old generation and decrease the size of young generation. Since survivor spaces are sub-partitions of the young generation, increasing the parameter newRatio will decrease the size of young generation and thus the size of survivor regions. In this set of experiments, we use h2, derby, and serial benchmarks with heap of 256MB, and compiler.compiler benchmark with heap of 128MB. Figure 2.9a shows that increasing newRatio value from 1 to 21 improves the runtime performance of application by reducing GC overhead. Figure2.9b, Figure2.9c, and Figure2.9d show that GC overhead is lower (i) for derby benchmark when newRatio is 8, (ii) for serial benchmark when newRatio is 3, and (iii) for compiler.compiler when newRatio is 8. Table 2.3 shows the number of Minor GCs and Full GCs, the time spent on Minor GC and Full GC with varying newRatio values for derby benchmark with heap of 256MB. We observe that as Minor GC overhead increases, Full GC overhead will decrease, as we increase the parameter newRatio value. However, the total GC overhead decreases until the newRatio value reaches 8 for derby benchmark. When newRatio value increases to 13 or higher, the total GC overhead increases, showing that there exists a newRatio value that minimizes total GC overhead. This set of experiments shows that depending on the application behavior and lifetime of the objects, newRatio value can be set appropriately for achieving optimal JVM runtime performance.



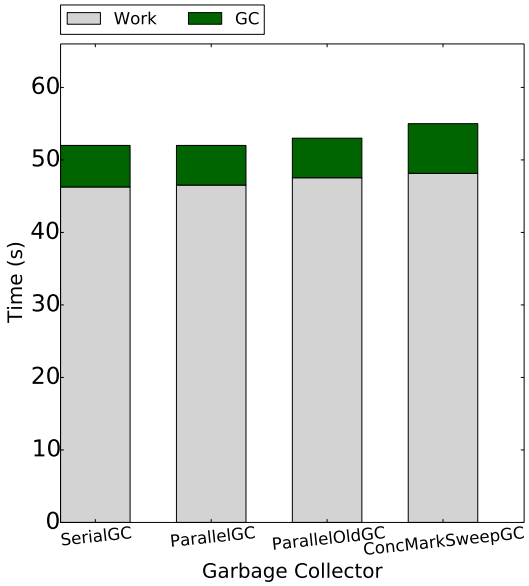
(a) h2

(b) derby

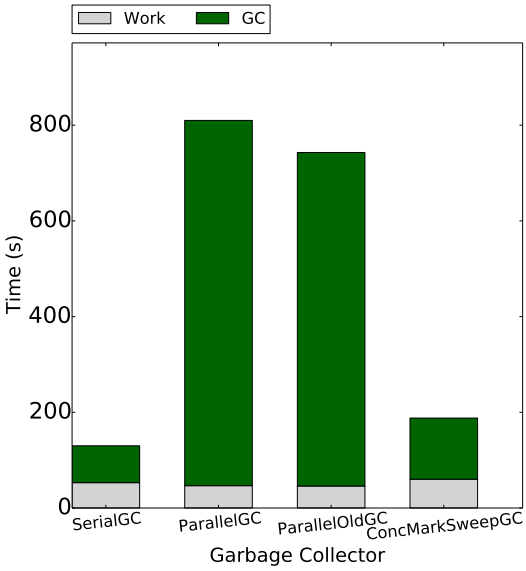
(c) serial

(d) compiler.compiler

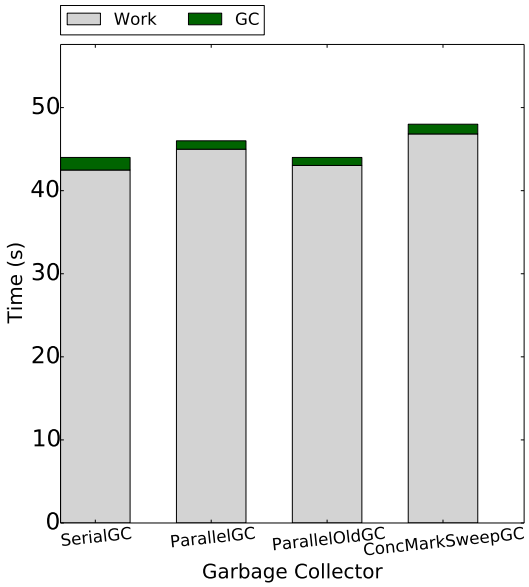
Figure 2.9: newRatio vs Running Time



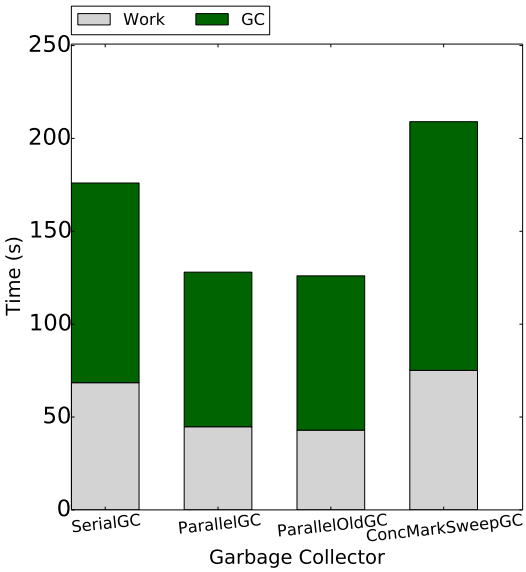
(a) h2 - 1024 MB



(b) h2 - 256 MB

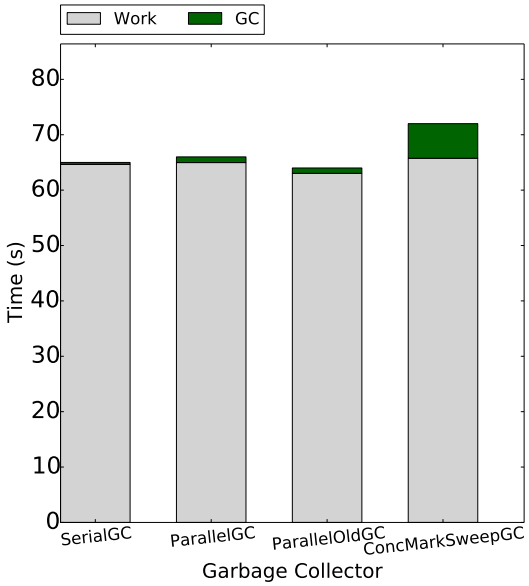


(c) derby - 1024 MB

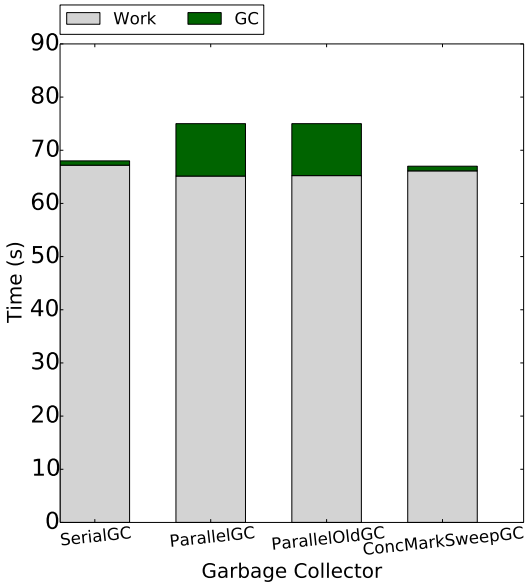


(d) derby - 256 MB

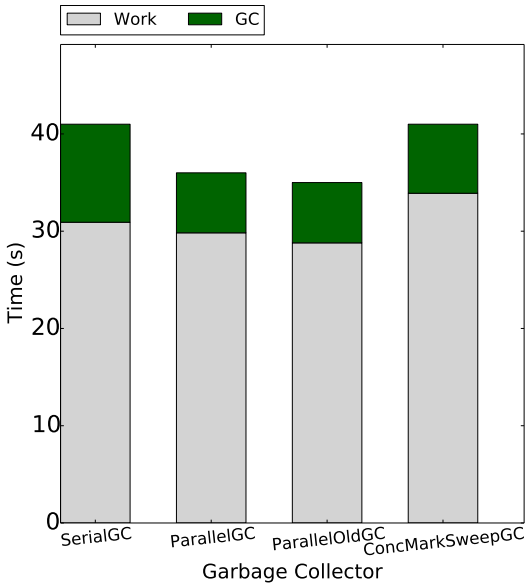
Figure 2.10: Garbage Collector vs Runtime (h2 and derby)



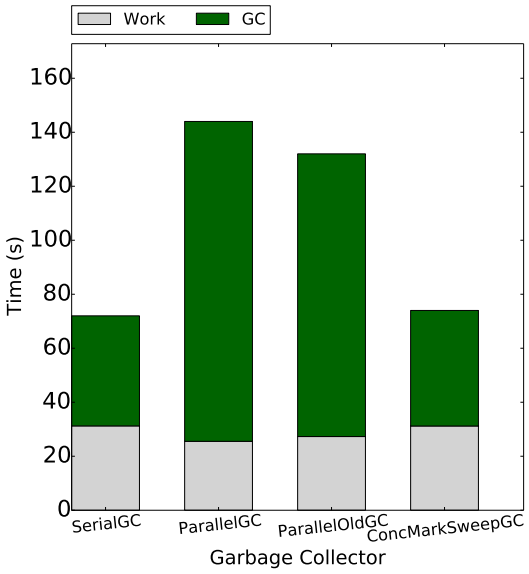
(a) serial - 1024 MB



(b) serial - 256 MB

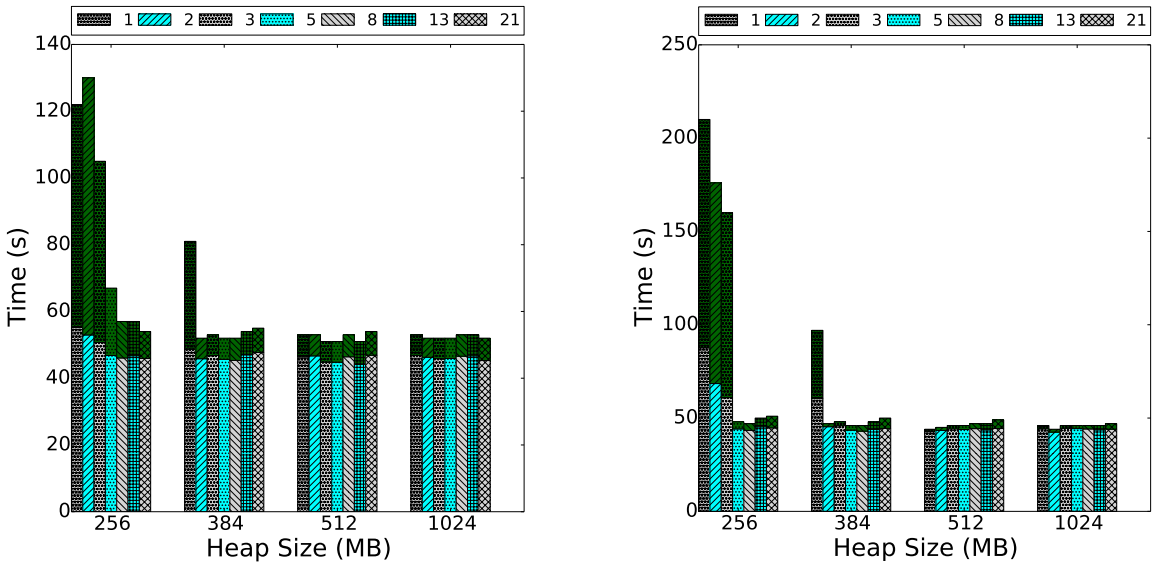


(c) compiler - 512 MB



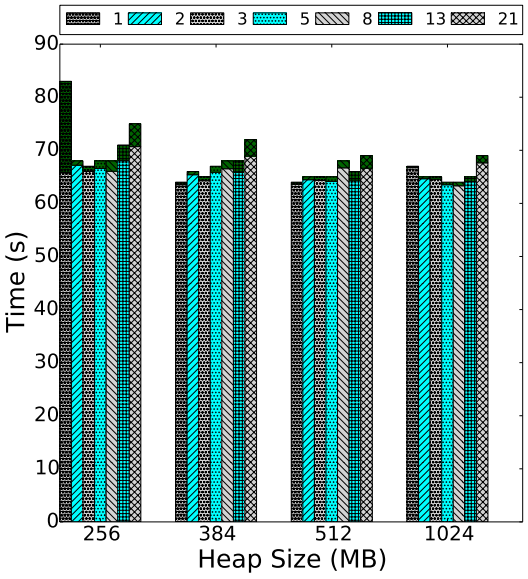
(d) compiler - 128 MB

Figure 2.11: Garbage Collector vs Runtime (serial and compiler)

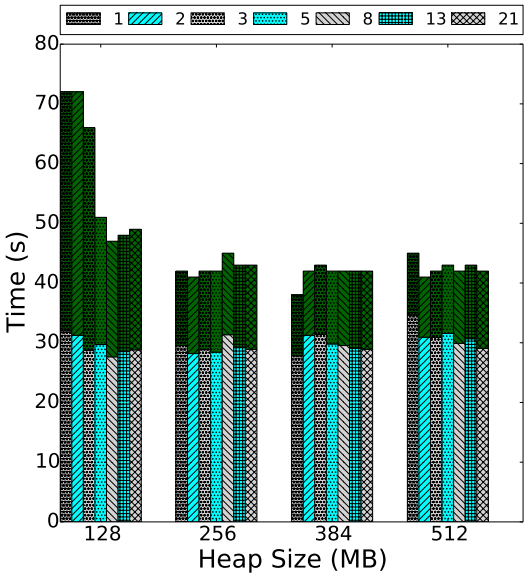


(a) h2

(b) derby

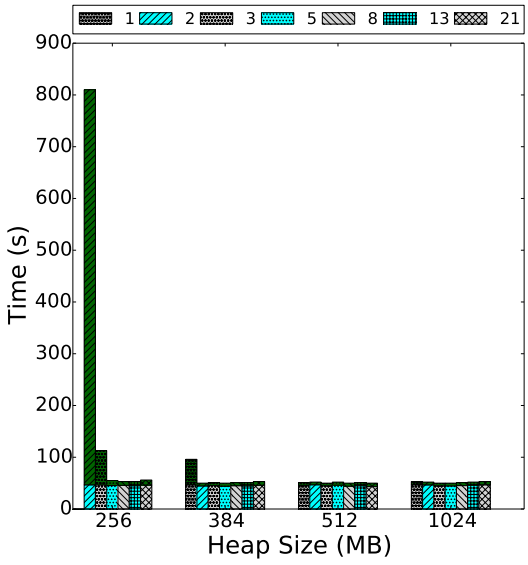


(c) serial

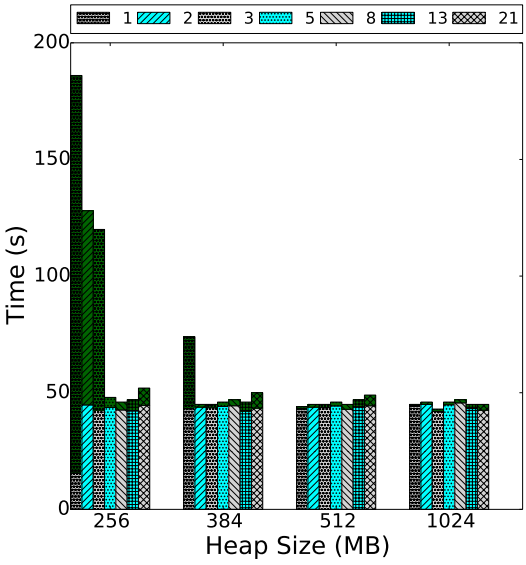


(d) compiler.compiler

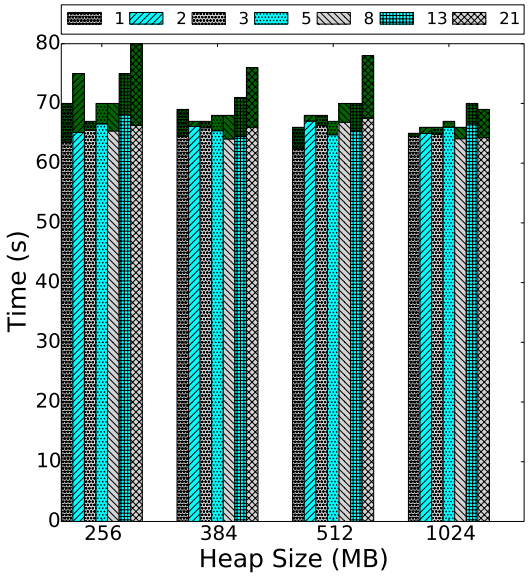
Figure 2.12: Heap Size & Structure vs Actual Work & GC Overhead (SerialGC)



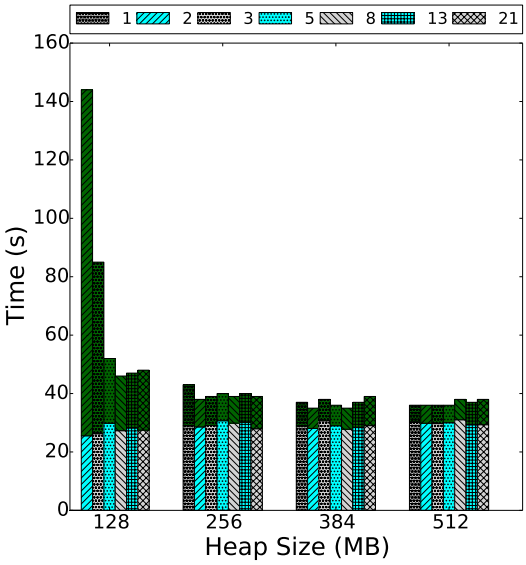
(a) h2



(b) derby



(c) serial



(d) compiler.compiler

Figure 2.13: Heap Size & Structure vs Actual Work & GC Overhead (ParallelGC)

2.5.4 Effects of Garbage Collectors

This section compares benchmark performance under varying garbage collectors. In this set of experiments, we run h2, derby and serial benchmarks with heap sizes of 1024MB and 256MB. We run compiler.compiler benchmark with heap size of 512MB and 128MB. Figure 2.10a and Figure 2.10b show that similar performance can be obtained when large heap is used for h2 benchmark. However, when heap is small, Serial and ConcMark-Sweep GCs outperform Parallel and ParallelOld GCs. Figure2.10c shows similar results for derby benchmark with large heap. However, in contrast to h2 benchmark, when heap is small, Figure 2.10d shows Parallel and ParallelOld GCs outperform Serial and Conc-MarkSweep GC. For compiler.compiler benchmark, Figure 2.11a and Figure 2.11b show that serial benchmark exhibits similar behavior as to h2. Figure2.11d shows that with small heap, Serial and Mark-Sweep GCs have lower GC Overhead and higher performance for compiler.compiler benchmark. Furthermore, for compiler.compiler workload, performance may differ even with larger heaps as shown in Figure 2.11c. In summary, the performance of collectors may vary with respect to different heap size and different application behavior. Additionally, heap space error can be a result of poor selection of garbage collector or poor setting of the newRatio parameter in addition to out of heap memory. Note that eden and survivor regions are fixed in size for Serial GC and ConcMarkSweep GC, but they are dynamically changing in Parallel and ParallelOld GCs.

2.5.5 Effect of Heap Size on Applications

This section investigates whether GC overhead and actual work of an application are highly correlated or completely independent processes. Given that object allocation and data access are parts of application's actual work, we want to analyze the effect of heap size, newRatio values on the time spent for performing actual work. In this set of experiments, we run h2, derby, and serial benchmarks with heap size varying from 256, 384, 512, to 1024 MB, and run compiler.compiler benchmark with heap size varying from 128, 256,

384 to 512 MB. We also vary newRatio value from 1 to 21. Figure2.12a, Figure2.12b, Figure2.12c and Figure2.12d present results for h2, derby, serial and compiler.compiler benchmarks respectively. These figures show time spent on performing actual work for specific application is the same regardless of the heap size or heap structure. The reason is that the data access cost is fixed, since they are all regular memory access, and the cost of object allocation is fixed because of compaction preserving feature of JVM. Figure2.13 shows similar results when using ParallelGC.

2.6 Conclusion

We have studied the effects of tuning JVM heap structure parameters and garbage collection parameters on application performance, without requiring any JVM, guest OS, host OS or hypervisor level modification. Our extensive measurement study has shown a number of interesting observations: (1) Increasing heap size does not increase application performance after a certain point. (2) Heap space error does not necessarily indicate heap is full and all objects in the heap are alive and heap space errors can be resolved by tuning JVM parameters. (3) By tuning JVM heap structure and GC parameters, we can achieve the same application performance using smaller heap sizes. Most of our measurement results are not specific to any garbage collection algorithm or any specific garbage collector implementation. We conjectured that our results can help software developers of big data applications to achieve higher performances by better management and configuration of their JVM runtime.

CHAPTER 3

DAHI: A LIGHTWEIGHT CACHING AND MEMORY COORDINATION FRAMEWORK FOR JVM EXECUTORS

Resilient Distributed Datasets (RDDs) are a fundamental core and hallmark of Spark. This chapter studies the performance impact of RDD management on Spark applications from two perspectives: (1) We show how RDDs should be configured with care in order to optimize the performance of iterative machine learning workloads on Spark when main memory is sufficient for RDD caching. (2) We show that applications on Spark experience large performance deterioration, causing unbalanced memory utilizations and premature spilling, when RDD is too large to fit in memory. To address these problems, we present DAHI, a light-weight RDD optimizer. DAHI provides two enhancements to Spark: (i) using elastic executors, instead of fixed size JVM executors; and (ii) supporting coarser grained tasks and large size RDDs by enabling partial RDD caching. Extensive experiments on machine learning and graph processing benchmarks show that with DAHI, the performance of these applications on Spark improves up to 12.4x.

3.1 Introduction

Spark [55] is a popular big data analytics platform for memory intensive applications. The concept of Resilient Distributed Datasets (RDDs) [56] is a hallmark of Spark. An RDD is an *abstraction* for a collection of immutable objects, distributed over the cluster in partitions. RDDs can be created either from the data on the stable storage or by transformation from other RDDs through Spark operations (e.g., map, filter and join).

With RDD abstraction, Spark makes use of memory in two different ways, compared to conventional frameworks, such as Hadoop MapReduce [70], MPI [71]. First, RDDs to be generated/used between consecutive map (1-to-many) operations are kept in memory. Sec-

ond, RDDs to be reused can be explicitly cached in memory. These enhancements improve application performance by reducing disk access and minimizing interprocess communication between executors.

However, this makes Spark greedy on memory usage, and exhibits several problems as amount of data to be cached increases. First, recall that each Spark executor is a JVM instance, and caching occurs on JVM heap by default. Hence, as RDD to be cached gets larger, the garbage collection overhead also increases. Secondly, in each worker node, multiple multi-threaded JVMs are launched to run a Spark Job, in which every JVM instance is given fixed portion of available memory. This design prevents one executor from using idle memory of another executor for caching, and result in performance degradation in heterogeneous applications, or when skewness in data is observed. Third, caching is partition-wise operation, which means if there is an available memory to cache entire partition, it is performed successfully. Otherwise, it is spilled/discarded leaving memory underutilized. Although, having coarser partitions results in low scheduling overhead, high shuffle optimization and low network traffic, it may hurts overall performance by causing low memory utilization, and more spilling/discarding.

Existing research to address these problems can be analyzed in two set. The first set of studies focus on improving JVM's memory management, ranging from heap resizing policies [53], JVM ballooning [63] for memory coordination, estimating working set [14] and memory bloating [66] [14] [72] to address high GC overhead, configuring JVM on memory intensive applications [58]. However, none of these studies optimizations for RDD management in Spark thereby their effects on Spark applications are limited. Also some require either JVM or application modification.

The second set of existing studies relies on external caching systems. These are either in-memory key-value stores [73], [74] or in-memory file systems [75]. The high level structure for these systems are similar: one caching instance is launched in every cluster node, and several features such as durability, load balancing/migration, fault tolerance

are provided over those instances transparently to the client -which is Spark in this case-. And client performs read/write operations to these systems over network. Although better performance can be measured compared to Vanilla Spark by reducing GC overhead, and achieving high memory utilization, they introduce additional overheads resulting from their several key characteristics. First key component of these systems are to enable clients to share data among each other. In order to do that, each client should write/read their data over network to/from cache instances. This can be beneficial and required if writer, and the reader clients are different applications, and the cost of data transfer is inevitable. However, in Spark, scheduling is a locality aware decision, enabling that data generation and computation over the data is performed on the same client-executor-. Second characteristics is to provide load balancing among caching instances. To achieve that, data is migrated from one caching instance to another. In return, this brings another data transfer and copying overhead over network. Third feature that causes additional overhead is to support fault tolerance. The conventional approach in these systems is to keep replica of the data. If replica is in memory, this will prevent us to cache more data in memory. If replica is in disk, in case of recovery disk access cost will be added. However, in Spark, lineage information of each RDD is kept. So, the fault tolerance is enabled not by replication, but re-computation.

In this chapter we present DAHI , as a light-weight caching and memory coordination mechanism for JVM applications, together with it's integration with Spark framework. First, we motivate our problem showing that optimizing RDD performance is critical for iterative machine learning workloads on Spark when main memory is sufficient for RDD caching. Furthermore, we show the adverse effect of RDD caching on latency-demanding big data applications when their RDDs become too large to fit in memory and why these applications are unable to leverage otherwise unused memory even when temporal memory imbalance and usage variations are observed across executors, for instance, one executor starts spilling, while another has plenty of idle memory. Second, we present overview

of DAHI for JVM applications focusing data caching, and memory coordination. DAHI addresses the above problems via two primary components: a node manager that coordinates memory utilization of JVMs, and d-store, to cache data in native memory, attached to JVM instances. We then describe some potential improvement opportunities by presenting a light-weight RDD optimizer with Spark-DAHI integration. DAHI improves Spark RDD managements from two aspects: unbalanced memory utilizations, and inefficiency in dealing with increased RDD caching demand and high garbage collection overheads. It enables Spark to have elastic executors, instead of having fixed size JVM executors, and it enables partial RDD caching on Spark via d-store, achieving high memory utilization while supporting coarser grained tasks and large size RDDs. Moreover, DAHI does so without any modification to the underlying architecture, the OSes, and user applications. Extensive experiments on open benchmarks show that using DAHI , the performance of analytics applications on Spark improves from 1.1x to 12.4x.

3.2 Related Work

This chapter rethinks the benefits and adverse effects of RDD management in the context of Spark analytics applications. Although existing proposals have studied Spark Map-Reduce performance, ranging from heap resizing policies [53], JVM ballooning [63], estimating working set [14], memory bloating [66] [14] [72], and JVM configuration on memory intensive applications [58], they have not studied the optimizations for RDD management.

To improve Spark's performance, existing research has focused on task scheduling [57], enhancing data movement among executors over RDMA [76], improving shuffle performance [77, 78], minimizing RDDs memory consumption [72] or improving resource sharing among concurrent applications [16] [79]. However, existing proposals do not support elastic management of RDDs on demand.

3.3 Spark Overview

In this section, we are going to explain how Spark manages intermediate data, types of memory pressures created on the system, and storage layers for intermediate data.

Spark[55] introduces Resilient Distributed Datasets (RDDs) as its main data structure. An RDD represents the intermediate data and it is an immutable collection of read only objects. RDDs can be created from the data on the stable storage, or transformations (e.g. map, filter and join) applied on other RDDs. Since RDDs are immutable data structures, transformations applied to one RDD creates another RDD instead of modifying the content of the original RDD. To illustrate this, we use Logistic Regression application as an example in code 3.1. In line 2, we create a first RDD, *file*, from a text file using Spark API. To extract the points from *file*, we apply map transformation with a parsing filter, and store them in *points* RDD in line 5. Since *points* will be used in every iteration of regression, it is cached/persisted in memory in line 8. Finally, we apply certain number of iterations of map and reduce to compute gradient and update weights in lines 14-21. Every map, and reduce creates another RDD for each iterations of the application. We illustrate these RDDs created by the application in Figure 3.1.

Although this design, keeping RDDs immutable and creating one from another, has efficiencies and simplicity to recover from failures, it creates a great amount of memory pressure on the system since every version of the intermediate data occupies a certain amount of memory. As a recovery strategy, spark uses lineage graphs, in which lost RDD partition is rebuild by tracing the lineage information and reapplying the transformations from an existing RDD partitions. Having RDDs immutable enables a fast recovery since every versions of the intermediate data for each transformation may still exist, reducing the number of transformations applied to an existing RDD partitions and reducing the recovery time. However, this design means being greedy on memory consumption, and it creates a significant memory pressure on the system, and triggers more frequent garbage collection.

```
1  // Create input RDD from a text file
2  val file = spark.textFile(...)
3
4  // Parse input RDD to extract points
5  val points = file.map(parsePoint)
6
7  // Cache points RDD because it will be used multiple times
8  points.cache() // points.persist(MEMORY_ONLY)
9
10 // Initialize weigths to random D-dimensional vector
11 var weigths = Vector.random(D)
12
13 // To update weights, run multiple iterations
14 for (i <- 1 to ITERATIONS) {
15       val gradient = points.map { point =>
16             point.features * (1 / (1 + exp(-point.label
17                   * weights.dot(point.features))) - 1) * point.label
18       }.reduce(_ + _)
19
20       weights -= gradient
21 }
```

Listing 3.1: Logistic Regession Application

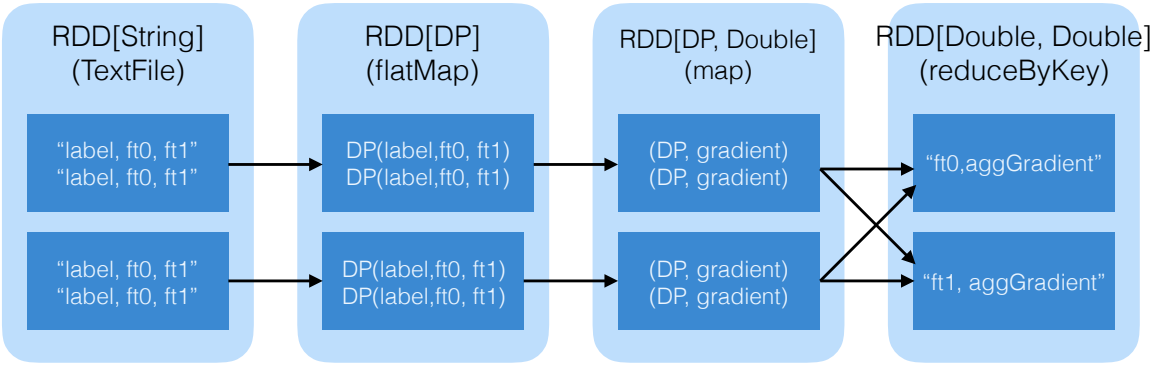
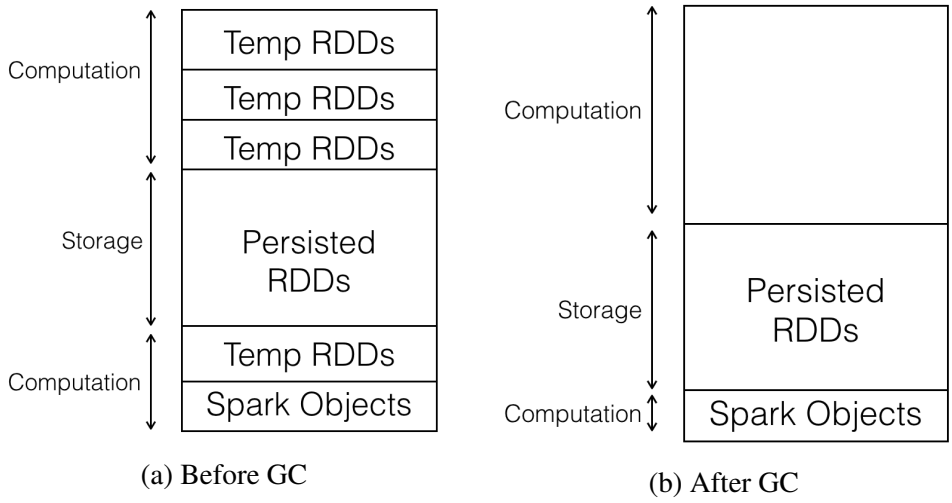


Figure 3.1: RDDs in Logistic Regression

Spark also allows its users to explicitly specify which RDDs are going to be persisted as we have seen in line 8 in Listing 3.1. The idea is that if an RDD is going to be used more than once, it should be persisted so that it can be accessed directly instead of computed again. Additionally, when persisted RDD will be no longer needed, user again needs to unpersist that RDD explicitly. There are different types of layers -a.k.a storage levels- to persist an RDD . We are going to evaluate each level in Section.... However, at this point we should note that when RDD is persisted in memory, this will also create a memory pressure, and increase garbage collection overhead.



(a) Before GC

(b) After GC

Figure 3.2: Spark Executor Heap

3.3.1 Temporary and Persisted RDDs

Executor memory in Spark is divided into two as for computation, and for persistency. Hence, we can divide the memory pressure caused by RDDs into two with respect to those:

- *Short Term Memory Pressure:* RDDs are temporary data structures unless they are explicitly persisted. As a result, each transformation creates another RDD throughout the task, but only the final RDD in the task matters as either shuffle data, or action result. At the end of the task, RDDs occupies a space in executor memory as dangling set of objects and stays there until garbage collection. We call these type of pressure created on the memory *short term memory pressure*.

- *Long Term Memory Pressure:* When user decides to persist an RDD in memory, Spark stores the pointers to the RDD partitions in order not to lose the data during garbage collection. When user unpersist that RDD, its partiton pointers are removed, so that RDD can become collectible during GC. Since lifetime of the persisted RDD is explicitly decided by user, and it can outlive/survive many garbage collection we call this type of pressure created on the memory *long term memory pressure*.

Together with executor memory, RDDs can be persisted in different layers. In the next section, we are going to evaluate each layer with respect to their capacities, garbage collection overheads, and read/write speeds.

3.3.2 Garbage Collection

In this section, we are going to overview the components of garbage collection overhead and re-formalize it in Spark context. In garbage collected languages, the execution environment is responsible for the entire memory management. Objects are created with keyword *new*, and during garbage collection memory occupied by dangling objects, the objects that cannot be reached from the application, is reclaimed back to the system. Garbage collection is triggered when the JVM heap is full, and application is suspended from execution during GC. Thus, it brings an overhead to the execution.

Garbage Collection overhead consists of three components [58]: frequency of heap getting full, tracing the JVM heap to identify which objects are dangling, moving alive objects contagiously to provide compaction for fast object allocation. Figure 3.2 illustrates the state of executor JVM before and after garbage collection. It shows that overhead includes tracing and moving cost of persisted RDDs together with the cost of temporary RDDs. To avoid this, RDDs can be persisted in different layers. In the next section, we are going to evaluate each layer in terms of their capacities, garbage collection overheads, and RDD read/write speeds.

3.3.3 Storage Layers

In this section, we are going to analyze each storage layer for RDDs to be persisted in Spark. Our metrics here will be storage capacity, garbage collection overhead, and RDD read/write speed. Before this evaluation, we should not that Spark does not take any action to provide load balancing for RDD storage. This means that if one executor is assigned more tasks than others, then it is going to compute partitions of an RDD and store more

Table 3.1: Storage Layers

Storage Layer	Capacity	GC Overhead	Read/Write Overhead
Executor Memory	Memory	High	Memory
Disk	Disk	Low	Disk
Ramdisk	Memory	Low	Memory + Disk I/O
Off-heap	Memory	Low	Memory + Network

partitions of that persisted RDD. When an executor needs a partition that it does not have, then it receives that partition by communicating with the executor that computes it. Consequently, we can say that load balancing in storage is out of Spark's scope, and that is why we do not consider in our storage layer comparison.

Executor Memory is the first and default storage layer for persistence in spark. If no storage layer is specified to persist an RDD. The capacity of the storage is a fraction of the executor memory, and it can be configured with *spark.memory.storageFraction* parameter. Therefore, it is bounded by the executors memory. As for our second metric, garbage collection overhead, persisting RDDs in the executor memory creates a significant amount of GC overhead. The reason is that persisted RDDs occupy certain amount of memory, causing executor memory -JVM heap- to get full faster. Moreover, since they will reside in the memory until they are unpersisted, those RDDs will not be collected during GC. For the speed of read/write time, this option gives the best performance as it is expected because RDD operations are done in memory speed.

Disk is the second storage layer to persist an RDD. We want to analyze this option into 2 parts: true disk and ramdisk based storage layers. **True Disk** is the storage layer with the maximum capacity. If RDD is too big to fit into the memory, and its computation time takes certain amount of time, one can benefit from this option to persist RDDs. And for the garbage collection overhead, since we detached storage from executor memory, it will not create a memory pressure and reduce the garbage collection overhead. For the

speed of read/write time however, persisting an RDD is the slowest option. Nonetheless, if re-computation of an RDD is more costly than reading from Disk, then this layer is still useful. As a second disk based solution, Spark user can create a **Ramdisk** and configure the system to persist RDDs on it through the same interface of Disk storage layer. The capacity of the Ramdisk is however bounded by the memory in the machine. The benefit of this option presents itself in the garbage collection overhead. Since storage is detached from the executor memory, we will experience lower garbage collection overhead. For our third metric, RDD read/write speed, our experiments showed that performance of Ramdisk is faster than Disk, but slower than Executor Memory. We believe the reason behind this is that even if RDDs still reside in the machine memory, we access them through Disk I/O. Hence, it brings another overhead that does not exist in accessing the RDD in executor memory.

Off-heap is the last storage layer to persist an RDD. Even though Tachyon is the only service that is used by Spark, we include in-memory key-value stores in this layer. When off-heap is selected, Spark sends the persisted RDD partitions to another system, e.g Tachyon, Redis, AsterixDB etc, using their clients. The capacity of this storage layer is limited to the memory as well. For our second comparison metric, detaching storage from executors' memory reduces the garbage collection overhead. In read/write speed of an RDD, although off-heap storege options keep the RDD in-memory, they have an additional overhead caused by the network stack since it is the only method for the communication between Spark executors and off-heap clients.

We summarize this evaluation in Table 3.1 and it shows that even though memory based storage layers have limited capacity and they present additional overhead -e.g GC Overhead for Executor Memory, Disk I/O Overhead for Ramdisk, and Network Overhead for Off-heap-, they outperform Disk in terms of read/write performance. The second observation is that GC overhead caused by persisted RDDs can be eliminated by detaching Storage Layer

from Executors' JVM heap.

RDD-Depot which manages persisted RDDs in native code thorough Java Native Interface (JNI) addresses the aforementioned problems by constructing a executors memory is a combination of JVM heap and native memory. While temporary RDDs are kept in JVM heap, persisted RDDs are stored in native memory. Similar to Disk and Off-heap storage layers, this design also detach storage and computation memory. Moreover, since RDD-Depot is a library plugged in to the executors, read/write is in-memory speed without any additional overhead. In Section...., we are going to explain how RDD-Depot works and design goals behind it.

3.4 RDDs: Benefits and Adverse Effect

In this section, we describe the main benefits Spark's RDDs and analyze some adverse effect of RDD management. We use Linear Regression (LR) as an example to give an overview of RDD management and cost-benefit analysis.

LR is an iterative machine learning application that generates the prediction model from multi-featured input data points. First, it loads the input data points and *explicitly* caches them in memory. After creating the RDD for loading its input dataset, each iteration consists of subsequent map and reduce operations to compute gradient and update the model weight parameters.

No RDD caching. This is enforced by Spark runtime by default. Every time when an RDD is needed, it is computed from its parent RDD, which is in turn computed from the grandparent, and so on. In order to improve performance of iterative applications, application developers are encouraged to explicitly indicate which and when an RDD should be cached and uncached. Our experiments show that for Linear Regression with 10M (million) data points, when the working set can be fully cached in memory, the performance improvement is 56x with explicit RDD caching, compared to the **no-RDD-caching** default.

Partition: a unit of computation and caching. This is the second design choice in

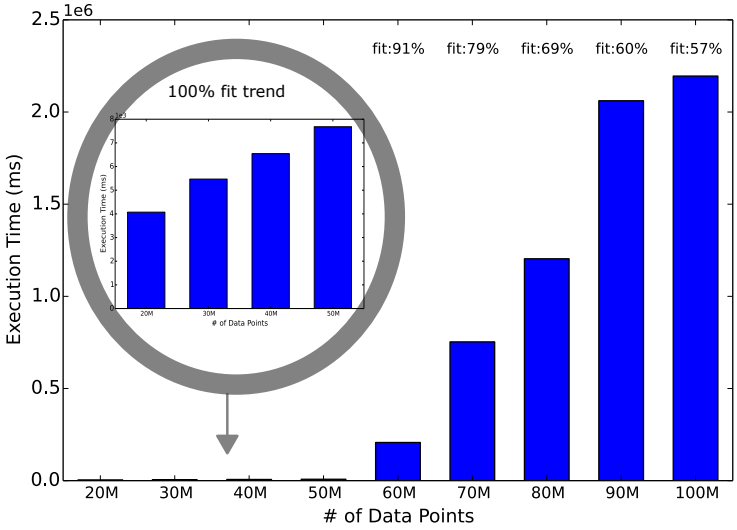


Figure 3.3: LR Average Iteration Time w.r.t. varying input sizes

RDD management. In Spark, partition and task granularity are used interchangeably. Data parallelism is utilized in Spark through RDD partitions by enforcing concurrent executions of multiple tasks, with each task performing its computation on one of the many partitions of an RDD. Similar to computation, RDD caching is also performed in the unit of partition. That is, a partition can only be cached, if it fully fits into its executor's memory.

Partition Re-computation v.s. Spilling to Disk. The decision is made regarding whether to perform partition re-computation or spilling the partition to disk when memory is insufficient to cache every partition of an RDD. By default, missing partitions are recomputed each time when they are needed. To deal with the situation in which RDD partitions are too expensive to be recomputed, Spark offers an alternative option by spilling those partitions that cannot fit into the cache memory to disk. In the spilling option, if RDD partitions are generated for the first time, then serializing partition first before writing it to disk can help minimize the latency of disk accesses. Thus, the overhead of serializing and writing to disk is added. Similarly, in the subsequent stages, reading from disk and partition deserialization are required to bringing missing partitions back to executors memory. Depending on the system specification, and application behavior, one might choose one op-

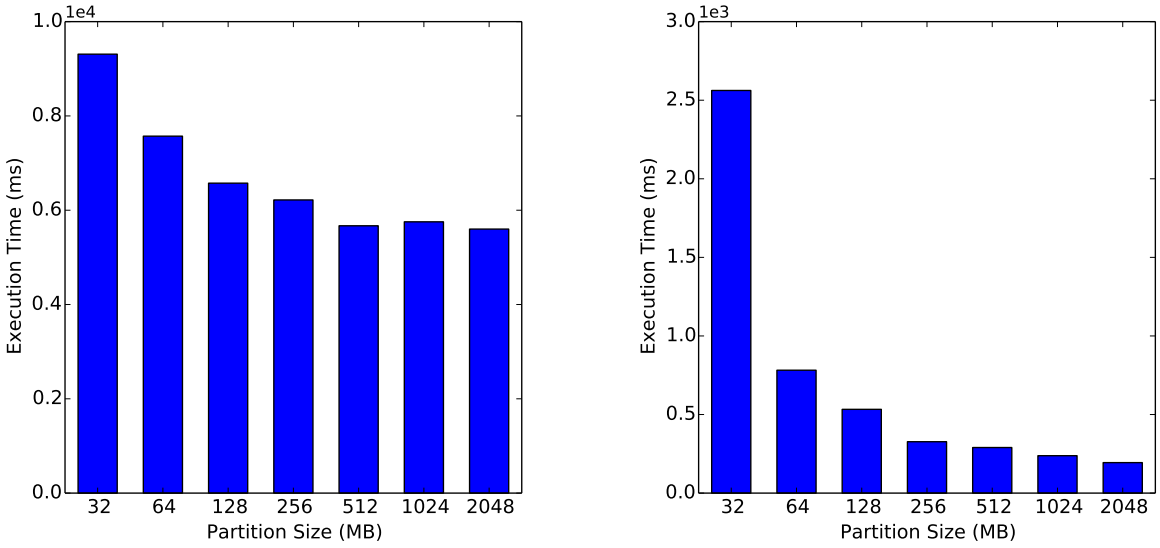


Figure 3.4: LR completion time w.r.t. varying partition sizes

tion over the other. Yet, it is important to note that Spark's performance drops significantly when its executors do not have sufficient memory for RDD caching.

To illustrate such adverse effect of RDD caching in the presence of larger RDDs, we run Linear Regression using 20M to 100M data points, increasing input data by 10M data points in each run. Figure 3.3 shows that there is a linear relationship between average iteration time and input size, as the input size is increased from 20M to 50M data points, because 100% of RDD partitions can fit in memory, enjoying full benefit of RDD caching. However, starting from 60M data points, the linear trend disappears and LR performance drops by 27x for input size of 60M, when 91% RDD partitions fit in memory and only 9% of RDD partitions cannot be cached. For the input size reaches 100M, the LR performance drops by 286x, when only 43% of RDD partitions cannot be cached.

Partition granularity: This parameter also plays a crucial role on Spark performance, due to its effects on CPU utilization, scheduling, and shuffle overheads. While fine grained tasks can induce high scheduling overhead, with coarse grained tasks, some executors/cores may stay idle, preventing system from achieving full CPU utilization. In addition, each map task performs local aggregation (map-side-combine) on the RDD partition, which will be used in the next reduce stage. Thus, using coarse grained tasks, the system can

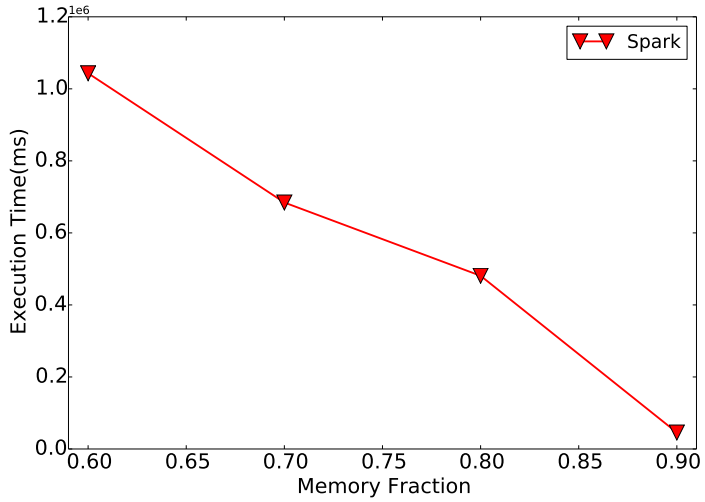


Figure 3.5: LR Completion time w.r.t. varying memory fraction

more effectively utilize the local aggregation, which in turn speeds up the reduce stage computation by decreasing the amount of shuffle data to be transferred among executors.

To show the effect of partition granularity, we measure the execution time of map and reduce stages of LR separately using different split sizes, varying from 32MB to 2GB , doubling in every run. Figure 3.4 shows that by having coarse grained tasks, we can reduce scheduling overhead and improve the performance of map stages. Similarly, the utilization of map-side-combine with coarse grained tasks improves the reduce stage performance. Intuitively, for both map and reduce stages, such improvement will converge eventually, and at some point, the performance will start to degrade, resulting from having idle executors.

Memory fraction. The fraction of executor memory to be used for RDD caching can be configured in Spark. This value is set to 0.6 by default, which is chosen as a threshold to avoid high garbage collection overhead and possible out of memory (OOM) error. On the other hand, by increasing memory fraction, one can cache more RDD partitions. Consequently, recomputation/spilling overhead is reduced, and overall performance can be improved. Figure 3.5 measures the time of average iteration of LR with 80M data points, using different memory fractions from 0.6 to 0.9, increasing by 0.1 in each run. When memory fraction is 0.6, only 69% of RDD partitions for this LR workload fits in memory and thus can be cached. The results show that increasing memory fraction improves LR

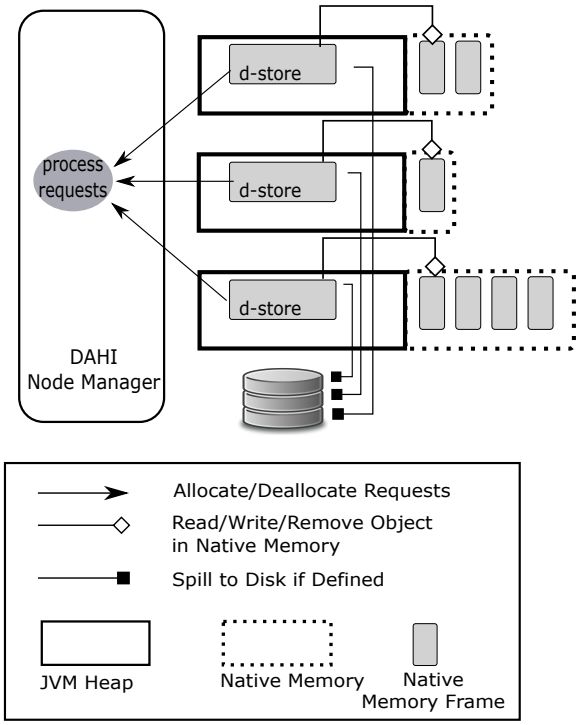


Figure 3.6: Caching and Memory Coordination with DAHI

performance. Although garbage collection overhead takes 38% of total execution time, when fraction is set to 0.9, application performance is improved by 23x.

3.5 Improving RDD Caching with DAHI

In this section, we describe three techniques for improving memory efficiency for RDD caching and runtime performance of Spark applications, when the RDD working sets do not fit into their executors memory. We implement these techniques in DAHI , a light-weight RDD optimizer.

Elastic Executors. There is no coordination for memory utilization among Spark executors, as each one is launched as an independent, fixed size JVM executor. However, partitions in the same RDD can be in different sizes from application to application. Thus, fixed-size executors have to work with variable size RDD partitions depending on application-specific requirements. As a result, when one executor's memory is overloaded, another executor can be underutilized. Eventually, having fixed size executor design to

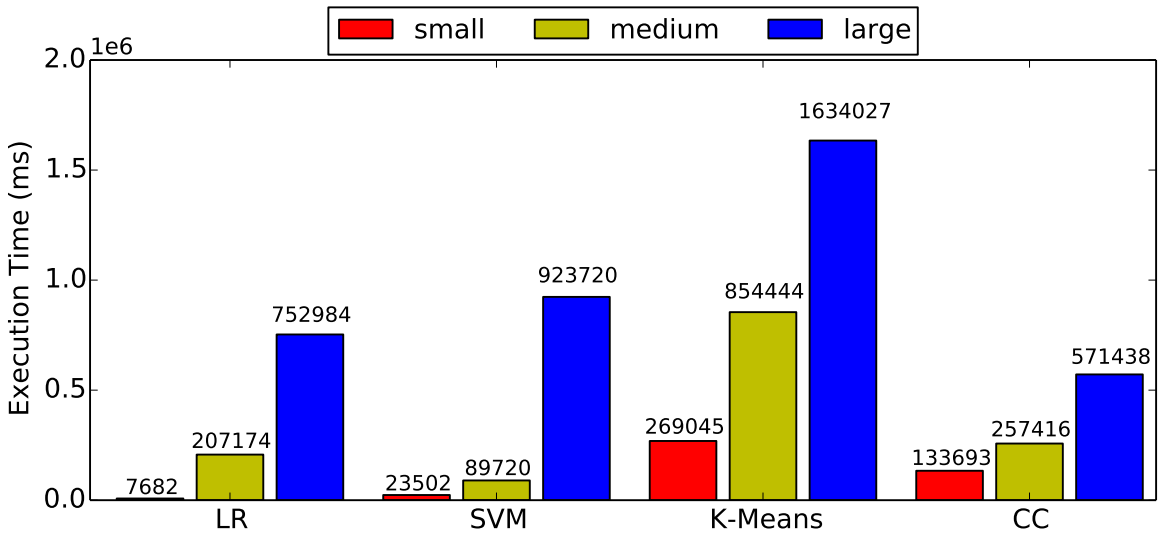
work with variable size RDDs and partitions is not scalable, and it prevents the executors, which lack memory to cache their working set of partitions, from leveraging unused memory in other executors. At the same time, applications exhibit low partition caching ratio, even when there is a plenty of unused memory.

One idea is to make executors elastic instead of fixed-sized. To validate the effectiveness of this idea on Spark executors performance, we implement it in DAHI by launching a node memory manager in each node, which orchestrates memory utilization in cache operations of every executor on the node. Instead of dividing available memory into n number of executors, each having JVM heap of size m and uses p fraction of heap for RDD caching, DAHI launches n executors, each having inital JVM heap of size $m * (1 - p)$, and having the rest of memory of size $nxmxp$ managed by node manager. Upon partition caching, executors increase their memory utilizations via the corresponding node managers' coordination. The communication and caching are performed via d-store instances in executors. Similarly, when a partition is evicted/removed from cache (i.e., uncached), the node manager is notified to allow executor shrink its memory footprint. This provides elasticity in executors, which enables a memory-overloaded executor to allocate more memory than the others on demand, and improves overall performance by enabling executors to leverage unused memory more efficiently.

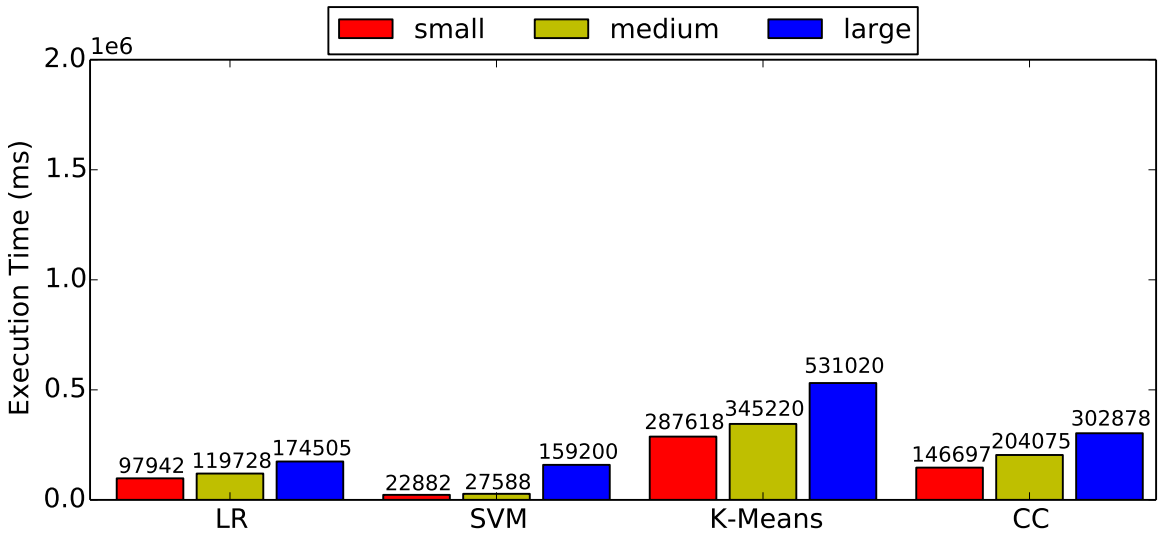
Light weight partition caching with d-Store. Recall Section 3.4, increasing memory fraction can lead to higher RDD caching ratio, which may result in performance improvement of applications as well as increased GC overhead (Figure 3). This is because RDD partitions are usually long-lived, possibly large objects, and as memory fraction for RDD caching increases, the system also exhibits high garbage collection overhead. At the same time, increasing memory fraction ratio for RDD caching does not address the utilization of unused memory across executors. This motivates the idea of creating d-store instances associated to executors on a node as light-weight partition cache by leveraging off-heap space. This can also help reducing GC overhead due to increased memory fraction for ex-

ecutors RDD cache. DAHI creates and attaches its d-store instances with executors on a node, which communicate and coordinate with its node manager. It is also responsible for expanding/shrinking executor's memory and caching/uncaching partition. Through Java Native Interface (JNI) calls, d-store can selectively and opportunistically moves some materialized partitions, (i.e., partitions fully residing in on-heap RDD cache) off-heap, and brings them back when they are needed by the applications. This off-heap cache is light weight, elastic and can effectively reduce GC overhead induced by long-lived, large RDD partitions, resulting in increased system efficiency in memory utilization and improved performance of executors and applications. DAHI does so without any modification to the OSes, and user application programs running in executors.

Partial Partition Caching. Partition is used as the unit of caching in Spark. In other words, if available memory is less than the size of a partition, the entire partition is discarded, leaving some memory idle (unused). Given the sizes of RDDs and their partitions may vary notably depending on application types, workload inputs, data characteristics and computation patterns, this ideal memory might be a large portion of memory, depending on application behavior and partition granularity. Independent of recompute/spill preference, DAHI takes a proactive approach to increasing memory utilization non-intrusively and opportunistically, to minimize the number of partitions being discarded by preset RDD caching policy of executor. When the total of unused memory slabs granted by node manager to the executors d-store is larger than the partition size, the d-store makes the caching decision on this partition by using one of the following options: (1) caching the entire partition if the total of unused memory slabs after adding the size of this partition is still above a default but configurable threshold, say 60%; (2) caching a partial partition instead of the whole partition, for example, by dividing the partition into two parts such that the first part is cached, and the second part is spilled to disk. In Option (2) the order is maintained so that when this partition is needed, the d-store serves the two parts in the same order. In all experiments reported in this paper, DAHI uses the second option.



(a) Spark



(b) DAHI

Figure 3.7: Spark vs DAHI - Execution Time under Different Input Sizes

3.6 Evaluation

In this section, we present our experimental results. First, we perform base experiments, providing a high level performance comparison of Spark and DAHI under varying applications and input sizes. Second, we compare the effectiveness of two system in partition caching. Finally, we provide experiments showcasing DAHI 's improvement in iteration

time, and garbage collection overhead under varying memory fractions.

We conduct our experiments on a host with 96 GB of physical memory and 2.67GHz Intel Xeon processor, containing 8 cores with hyperthreading. In Spark configuration, we are using 8 executors, each using 1 core and 10 GB of memory. Unless otherwise stated, we keep memory fraction value at default 0.6, and input split size at 256 MB.

We cover applications from both machine learning and graph processing domains, and all of these applications performs different type of iterative algorithm on the RDDs. The details of the applications are as follows:

Linear Regression: is a commonly used prediction algorithm that tries to determine linear relation between multi-dimensional feature vector and outcome. By evaluating the correspondence of input and output data points, it generates a model to estimate the output for any given data point. Our experiments uses 3 different input sets, containing 50M, 60M and 70M data points, each having 100 number of features.

Support Vector Machines: is another learning model that associates features of data points with the outcome. We use the same input data as we use for Linear Regression here: 50M, 60M and 70M data points, each having 100 number of features.

K-Means: is a clustering algorithm that partitions data points into *k* clusters. It starts with *k* randomly chosen cluster representatives. In every iteration, data points are assigned to a cluster depending on the distance/similarity to the representatives, and then representatives are updated as centroids of the data points assigned to clusters.

Connected Components: is an application we select from graph processing domain. The goal is to identify vertex sets, having in each set there is a path between every pair of vertexes, and there is no path for vertexes from different sets. For graph generation, we also used SparkBench, with parameters $\mu = 5$, $\sigma = 2$ and 0.8M, 1.0M and 1.2M vertices.

Before presenting our results, we want to clarify the reasoning behind experimental setup with these 3 input sizes *(small, medium, large)*. For all of our applications, RDDs to be cached, generated using *small* input, can fully fit into the memory. On the other hand,

for *medium* and *large* input sizes, spilling/recomputation occurs, since memory is not large enough to fit all RDD partitions. This setup also helps us to investigate SparkDAHI 's performance for both of the cases: where RDD can fully and partially fit into the memory.

3.6.1 Effect of Full and Partial RDD Caching

This set of experiments are performed on applications using 3 different categories of input datasets: small, medium, and large. For all applications, using *small* category dataset, the RDDs generated can be cached fully in memory, while using *medium* and *large* category datasets exhibit partial caching, as some partitions of the RDDs do not fit in memory. For LR, SVM, and K-Means, datasets of 50M, 60M and 70M data points, each having 100 features are used for small, medium and large categories respectively. SparkBench is also used to generate datasets of input graphs for CC, with parameters $\mu = 5$, $\sigma = 2$ and 0.8M, 1.0M, and 1.2M vertexes for small, medium and large categories respectively.

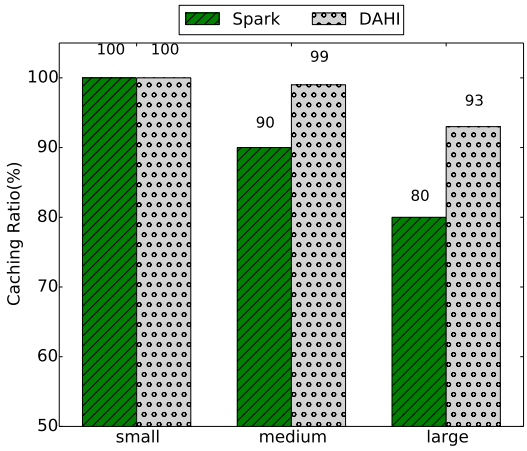
Full RDD Caching Performance. This set of experiments compare performance of DAHI with vanilla Spark for all four applications, when the RDDs of application can fully fit in memory. Using *small* datasets, we measure average iteration time of each application. Figure 3.7 shows the results. When applications have sufficient memory for full RDD caching, comparing to vanilla Spark, the only overhead that DAHI introduces is the object deserialization overhead. Figure 3.7 confirms this analysis, showing that DAHI exhibits nearly the same performance with vanilla Spark, and the deserialization overhead is negligible for SVM, K-Means and CC. However, compared to vanilla Spark, using DAHI , the execution time of LR is slightly increased. The reason is simple: SVM, K-Means and CC are more compute intensive than LR, and thus, their computation costs are much higher than, and dominate, the object deserialization overhead.

Partial RDD Caching Performance. The second set of experiments evaluates and compares the effect of partial RDD caching on DAHI with vanilla Spark. Using medium and large datasets, we report the results in Figure 3.7. It is observed that, using DAHI ,

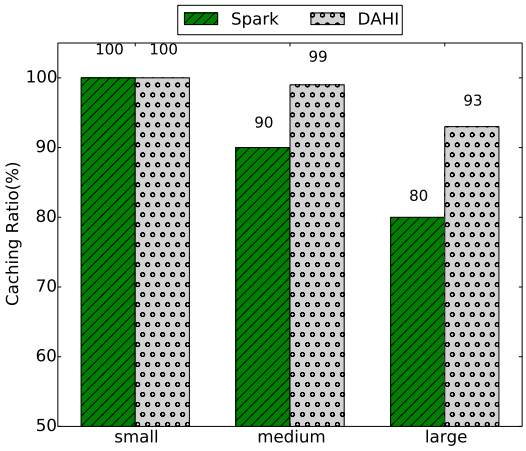
the completion time of LR obtains 1.7x and 4.3x speedup, over vanilla Spark, for *medium* and *large* datasets respectively. The speedup with respect to *medium* and *large* datasets for SVM is 3.3x and 5.8x, for K-Means is 2.5x and 3.1x, and for CC is 1.3x and 1.9x. The experiments confirm the analysis in Section 3. DAHI shows the benefit of exploiting unused memory sharing across executors through node level coordination via off-heap caching of those RDD partitions that cannot fit in their executors' memory.

3.6.2 Performance Impact of Caching Ratio

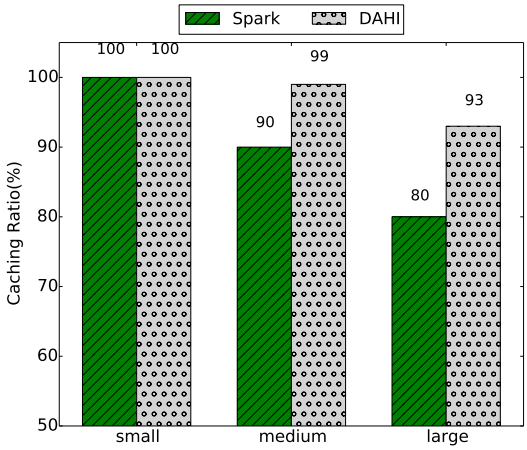
For a given application, the RDD caching ratio is defined as the portion of the cached partitions over the total number of partitions for all RDDs generated during the execution of this application. This metric is measured at runtime. Unlike RDD caching ratio, memory fraction is a configuration parameter associated to a Spark executor and it refers to the fraction of the total executor memory that is allocated to RDD caching. This set of experiments evaluates and compare DAHI with vanilla Spark on the impact of RDD caching ratio on the execution/completion time of applications. Figure 3.8 confirms that indeed using *small* datasets, 100% RDD caching can be achieved with both Spark and DAHI . However, when we increase the input dataset size to reach the *medium* category and then the *large* category, the caching ratio of Spark decreases between 5x and 20x, while the caching ratio of DAHI decreases up to 7x, for all four applications, compared to using *small* category datasets. Concretely, for LR, SVM, and K-Means, vanilla Spark can only cache 90% and 80% of RDD partitions with *medium* and *large* datasets respectively, whereas DAHI can cache 99% and 93% of RDD partitions in comparison. For CC, Spark can cache 95% and 91% of RDD partitions, when DAHI can cache 100% and 96% of the partitions. This experiment demonstrates from the caching ratio perspective the benefit of effectively orchestrating unused memory across executors: the RDD caching ratio is dropping slower as more RDD partitions cannot fit into executors memory by using DAHI , compared to vanilla Spark,



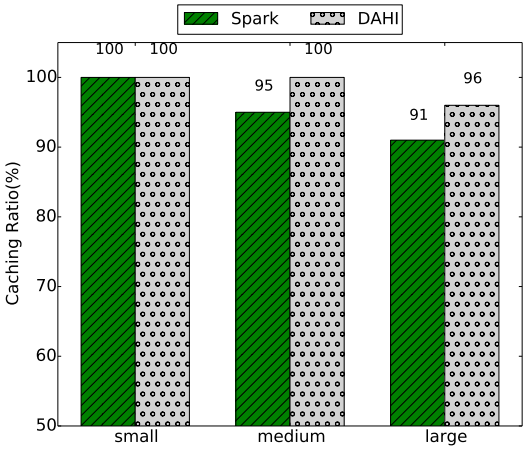
(a) Linear Regression



(b) Support Vector Machines

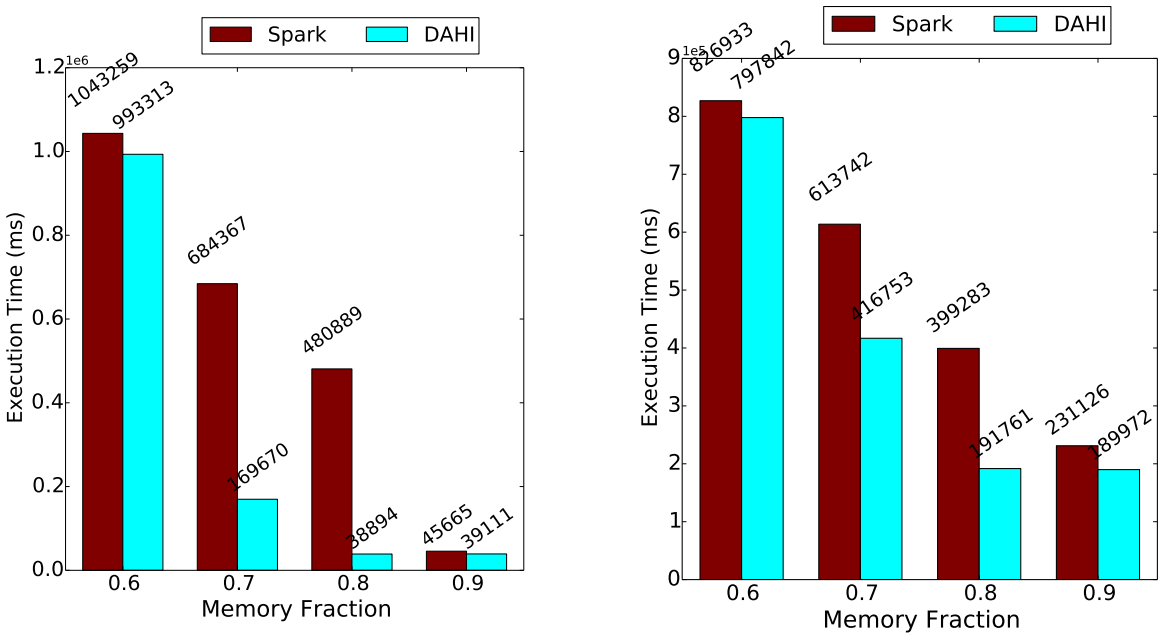


(c) K-Means



(d) Connected Components

Figure 3.8: Spark vs DAHI - Caching Ratio under Different Input Sizes



(a) LR

(b) CC

Figure 3.9: Spark vs DAHI - Execution Time under Different Memory Fractions

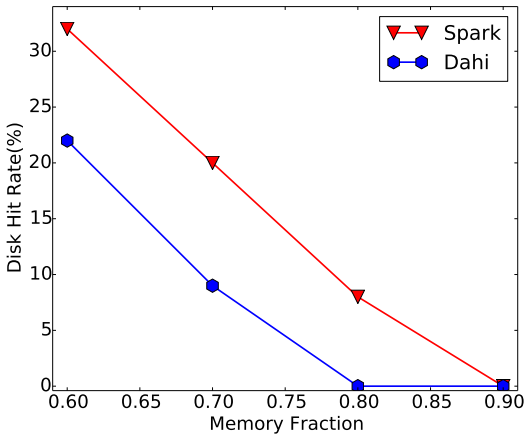
which further helps cutting down the overall execution time of applications.

3.6.3 Performance Impact of Memory Fraction

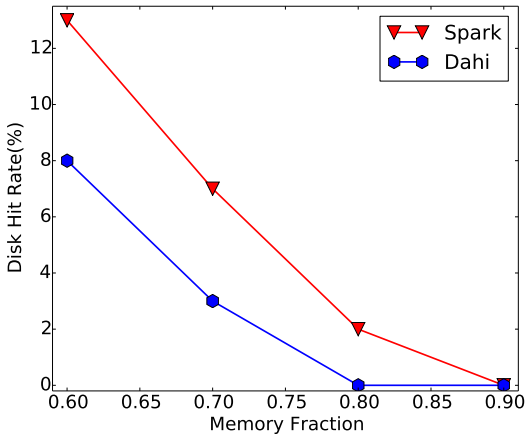
This set of experiments studies how different memory fractions impact the performance of applications when using DAHI , compared to using vanilla Spark.

We measure average iteration time of applications under varying memory fraction settings, from 0.6 to 0.9, incrementing by 0.1, on large category datasets. For LR, we use 80M data points with 100 features, and for CC, we use a graph generated by SparkBench with parameters $\mu = 5$, $\sigma = 2$ and 1.4M vertexes.

Figure 3.9 shows the results. We highlight two observations. First, for both DAHI and vanilla Spark, increasing memory fraction improves performance of applications by allowing more partitions to be cached in memory. With increased RDD cache ratio, disk accesses due to recomputation/spilling of cache-missed RDD partitions decrease, as shown in Figure 3.10. Second, as the memory fraction increases, using DAHI , execution time of both applications improves by 4.0x and 12.4x with memory fraction of 0.7 and 0.8 respectively

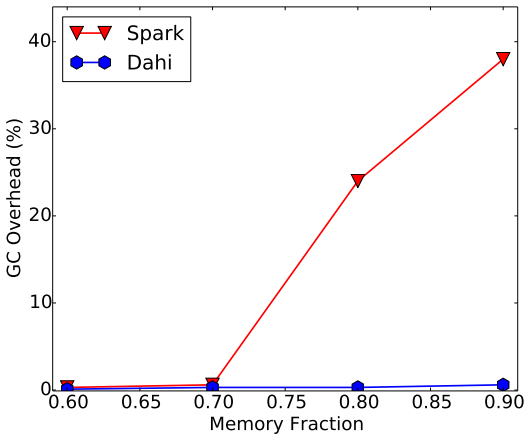


(a) LR

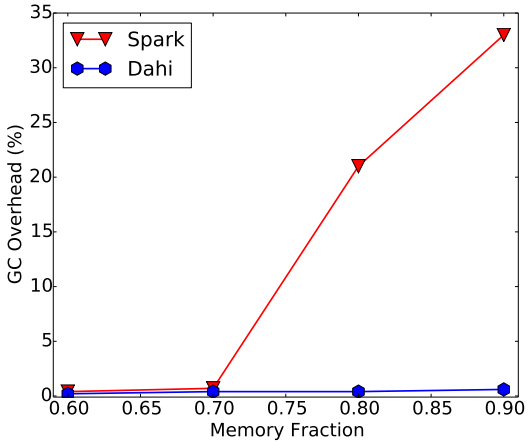


(b) CC

Figure 3.10: Spark vs DAHI - Disk Hit Rate under Different Memory Fractions



(a) LR



(b) CC

Figure 3.11: Spark vs DAHI - GC Overhead under Different Memory Fractions

for LR, over vanilla Spark. Even with memory fraction of 0.8, full RDD caching is not possible for LR under vanilla Spark. When memory fraction increases to 0.9, both system can cache all RDD partitions in memory, using DAHI , execution time of LR improves by 1.2x over vanilla Spark. Second, we observe similar trends in Connected Components. When memory fraction is 0.6, executions time of CC using DAHI is marginally better than vanilla Spark as DAHI caches 5% more RDD partitions than Spark. Increasing memory fraction to 0.7 and 0.8, DAHI improves by 1.5x and 2.1x over vanilla Spark, respectively, and has 1.2x speedup with 0.9 memory fraction, where both systems can cache all partitions in memory.

The reason that DAHI performs better than vanilla Spark even in the full RDD caching scenario is because RDD caching model in DAHI reduces GC overhead and increases RDD caching ratio. In comparison, for Spark, as memory fraction increases, not only the increase rate of its RDD caching ratio is lower than that in DAHI , as shown in Section 4.2, its GC overhead also increases.

Figure 3.11 displays the garbage collection overhead as a portion of total execution time. For LR, this overhead reaches 24% and 38% of the total execution time, as memory fraction is increased to 0.8 and 0.9 respectively. Similarly, for CC, this overhead goes up by 21% and 33%, as memory fraction is increased to 0.8 and 0.9 respectively.

3.7 Conclusion

We have studied the benefits and adverse effect of Spark RDD management. To address the problem of large performance deterioration when RDD partitions cannot fully fit in memory, we have presented DAHI, a light weight RDD optimizer with three enhancements: elastic executors, instead of fixed size JVM executors; partial RDD caching; and off-heap memory sharing for partition caching across executors on the same node. DAHI improves performance of Spark applications through enabling elastic executors and partial RDD caching. We have evaluated performance of DAHI comparing with vanilla Spark using machine learning and graph processing benchmarks, we have shown that with DAHI, the performance of these applications improves by up to 12.4x.

CHAPTER 4

DAHI-REMOTE

Today, many data centers have reported temporal imbalance of memory utilization across nodes in a cluster, including Google, Facebook. To address the potential problems of memory contention on some compute nodes in a cluster, we propose to extend DAHI development to provide DAHI-Remote capability. DAHI-Remote development is aimed to alleviate the high memory pressure of those executors that cannot find sufficient idle memory on their local nodes in a cluster by creating and providing remote memory sharing opportunities from other nodes in the cluster. The DAHI-Remote framework will enable memory coordination and caching among JVM instances across the entire cluster through a hierarchical RDD caching and memory sharing protocol. First, we enable JVM instances to have access their local native memory under local node and d-store management if available. Second, upon detection of insufficient memory for executors on the local node, DAHI-Remote creates and establish channels for remote native memory for RDD caching. For large size clusters, DAHI-Remote will create group based sharing such that each node can select and belong to one DAHI-Remote sharing group. The formation of DAHI-Remote sharing group is guided based on load balance, availability and overall performance of the applications on the cluster to ensure high throughput and low latency for applications and low garbage collection overhead for JVM instances, and efficient memory coordination among local and distributed executors across the cluster. DAHI-Remote also investigates policies for selecting remote cache node(s), selecting RDD partitioning schemes, aiming for high throughput, and fast data transfer over RDMA.

4.1 Motivation and Contributions

As the world becomes more instrumented and interconnected, the amount of data generated from software and hardware sensors increases exponentially. There are two main big data processing techniques to understand the data, extract correlations, and make predictions with high accuracy: 1) real time data processing [60] [80] [81] [82] [83], and 2) batch processing [59] [84] [85]. Spark [55] is a popular big data framework to achieve both of the goals. It implements Map-Reduce programming model in-memory, with its main abstraction of *Resilient Distributed Datasets (RDD)* [56]. RDD represents a collection of immutable objects, distributed over the cluster in partitions. They can be created either from the data on the stable storage or by transformation from other RDDs through Spark operations (e.g., map, filter and join).

Having extensive of use of memory, Spark outperforms its predecessor Map-Reduce frameworks in iterative jobs. In-memory computation is achieved in two different ways: 1) consecutive 1-to-1 map operations are fused together, and processing on single item goes through all map operations at once without spilling, or check-pointing in disk, and 2) it exposes an API to allow application developer to cache RDDs that will be known to be frequently used.

However, this makes Spark greedy on memory usage, and exhibits several problems as amount of data to be cached increases. First, recall that each Spark executor is a JVM instance, and caching occurs on JVM heap by default. Hence, as RDD to be cached gets larger, the garbage collection overhead also increases. Secondly, in each worker node, multiple multi-threaded JVMs are launched to run a Spark Job, in which every JVM instance is given fixed portion of available memory. This design prevents one executor from using idle memory of another executor for caching, and result in performance degradation in heterogeneous applications, or when data is skewed. Third, the primary goal of Spark scheduling is to load balance CPU utilization of nodes. Tasks are assigned ignoring availability of

memory for executors to cache data, and furthermore, executors in one node cannot utilize idle memory in another node, when it is under memory pressure.

Existing research to address these problems can be analyzed in three set. The first set of studies focus on improving JVM's memory management, ranging from heap resizing policies [53], JVM ballooning [63] for memory coordination, estimating working set [14] and memory bloating [66] [14] [72] [86] [87] to address high GC overhead, configuring JVM on memory intensive applications [58]. There are also studies that proposes shared memory system based JVM runtime in a single machine, and across nodes in cluster [88] [89] [90] [91]. However, these efforts remained in their prototyping stages, and neither themselves nor their proposed methods are currently available. Also, none of these studies optimizations for RDD management in Spark thereby their effects on Spark applications are limited.

The second set of existing studies focus on improving big data frameworks, by replacing their underlying TCP/IP based communication components, with RDMA [92] [93] [94] [95]. While these solutions promotes the use of RDMA in the communication among the nodes, they do not take holistic approach by combining memory optimizations, with communication.

The third set of existing studies relies on external caching systems. These are either in-memory key-value stores [73], [96] [97] [74] or in-memory storage systems [75] [98] [99]. The high level structure for these systems are similar: one caching instance is launched in every cluster node, and several features such as durability, load balancing/migration, fault tolerance are provided over those instances transparently to the client -which is Spark in this case-. And client performs read/write operations to these systems over network. Although better performance can be measured compared to Vanilla Spark by reducing GC overhead, and achieving high memory utilization, they introduce additional overheads resulting from their several key characteristics. First key component of these systems are to enable clients to share data among each other. In order to do that, each client should write/read their data

over network to/from cache instances. This can be beneficial and required if the writers, and the readers are different client applications, and the cost of data transfer is inevitable. However, in Spark, scheduling is a locality aware decision, enabling that data generation and computation over the data is performed on the same client-executor-. Second characteristics is to provide load balancing among caching instances. To achieve that, data is migrated from one caching instance to another. In return, this brings another data transfer and copying overhead over network. Third feature that causes additional overhead is to support fault tolerance. The conventional approach in these systems is to keep replica of the data. If replica is in memory, this will prevent us to cache more data in memory. If replica is in disk, in case of recovery disk access cost will be added. However, in Spark, lineage information of each RDD is kept. So, the fault tolerance is enabled not by replication, but re-computation.

In this chapter, we present DAHI-Remote as an extension of DAHI, a lightweight, distributed RDD caching mechanism together with its implementation in Spark. First, we motivate our problem explaining the details of Spark runtime, and how critical RDD caching is for the performance of iterative applications. We evaluate performance of Spark in two main categories, 1) where all RDD partitions can fit into the memory, and 2) where they cannot. Our analysis shows that the reasons behind the second case can either stem from scheduling decisions of the framework, or memory imbalance caused by application behavior. In section 4.2 overview of DAHI. Then in Section 4.4, we focus on transport layer added in DAHI-Remote to utilize idle memory in remote nodes in the cluster. We evaluated 3 different alternatives to implement RDD transport layer, including both TCP/IP and RDMA based solutions. In section 4.6, we discuss policies that shape remote node selection, secondary partitioning, RDD ownership update and fault tolerance decisions of DAHI-Remote. Finally, we evaluate performance of DAHI-Remote, by presenting our extensive results on selected benchmark applications from machine learning, and graph processing domains.

4.2 Overview of DAHI-Remote

DAHI-Remote is a mechanism for JVM applications to utilize GC immune caching and establish memory coordination among each other, both in the node, and across nodes. Although our implementation is for JVM applications, the design principles we discuss here can be extended to any other garbage collected infrastructures.

The main goal of DAHI-Remote is to have JVM applications to perform data caching with low GC overhead, and allow memory footprint elasticity providing coordination with other JVM applications running simultaneously on the same machine. DAHI-Remote achieves these, with it's two main components working together. The *d-store instances* that each participating application uses for registration and caching in both local memory, and remote memory, and *memory manager* that coordinates memory utilization in cache operations performed through *d-store instances* in the same node, and caches data coming from remote nodes.. Figure 4.1 illustrates how DAHI-Remote provides caching and memory coordination among JVM applications.

d-store instances are the objects owned by each participant JVM instance. They allow each participant to register itself to *memory manager* and perform caching operations. *d-store instances* provide an API to write/read serialized objects in streams from JVM heap to application's native memory first, if there is no available memory in the current node, streams to Node Manager in a remote node under *memory-manager*'s coordination. Write operation is only performed if permission to extend memory footprint is granted by *memory manager*. Node Manager gives that permission if there is an available local memory, or available remote memory in the cluster. Otherwise, the object to be cached is either written to disk, or discarded and application is notified. Similarly, when object will be removed from cache, *d-store instance* removes it from application's native memory, shrink its memory footprint and update *memory manager* on deallocation.

memory manager is launched for applications to register themselves, and perform

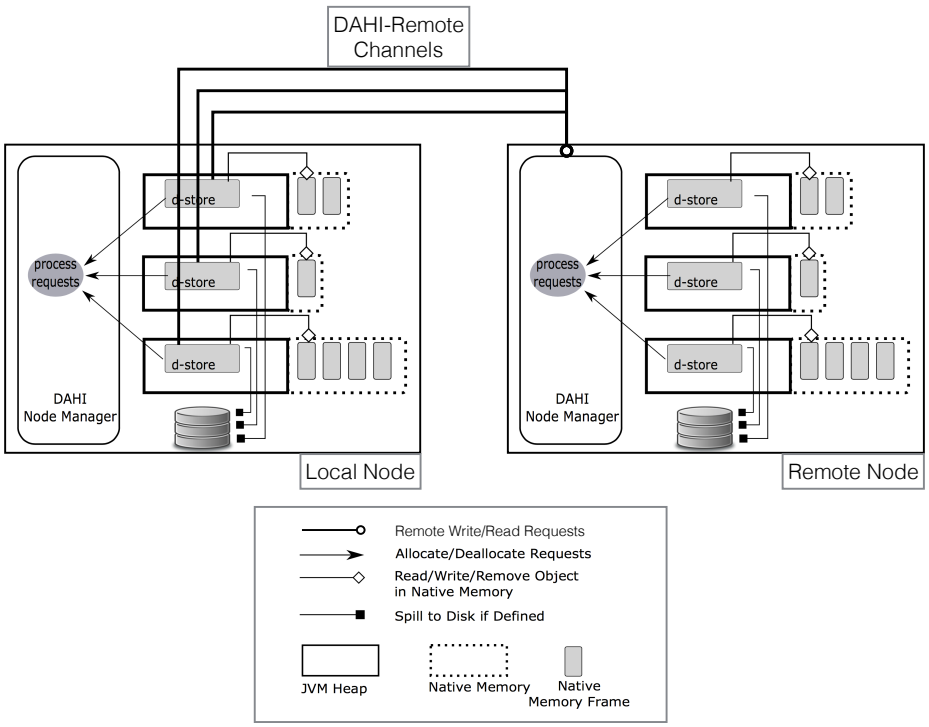


Figure 4.1: Caching and Memory Coordination with Across Nodes with DAHI-Remote

memory allocations and deallocations for caching operations in a coordinated way. We should note that *memory manager* is responsible for bookkeeping of memory operations of the executors in the local node, and caching for the executors in the remote node.

4.3 Advantages of DAHI-Remote

Utilizing memory in a coordinated way across participant applications, and across nodes, achieves high and controlled memory utilization, and brings elasticity to applications, addressing problems caused by non-uniform memory utilization, and premature spilling.

Low GC overhead is the first performance improvement that DAHI-Remote achieves. Garbage collection is triggered on the objects resides in JVM heap. However, by moving data to be cached to application's native memory or remote memory, DAHI-Remote improves overall application performance by reducing garbage collection overhead.

Fast data access can also be established via *d-store instances*, compared to external caching mechanisms (e.g key/value stores, in-memory file systems etc). In those solutions,

data is written/read to/from caching system requiring an inter-process communication (e.g tcp/ip). However, by keeping data in application's own native memory, we achieve faster data access with DAHI-Remote. Furthermore, exploiting remote memory achieves faster data access compared to spilling disk.

Elasticity in executors is achieved with DAHI-Remote, instead of statically launching them with fixed amount of memory. Total available memory is utilized among participant executors in the node. Each participant can extend it's own memory footprint on-demand with *memory manager*'s permission. This design allows memory intensive applications to utilize available memory better than others. Moreover, it does not exhibit problems caused by memory over-commitment (when total memory demand is higher than total available memory), because memory utilization of executors are controlled by node manager in DAHI-Remote.

4.4 Decisions for RDD Transport Layer

To be able to utilize idle remote memory in the cluster, DAHI needs to expand its architecture implementing a RDD transport layer over the network. Two main components of DAHI (node manager and d-store instances) are modified as follows:

Node Manager was responsible for coordinating memory allocation/deallocation requests coming from the executors in the node. While it is orchestrating these requests, data was cached in executors' native memory. However, in DAHI-Remote, our goal is for executor to utilize idle memory in remote nodes. To achieve this, caching capability is added in Node Manager. Together with coordinating memory requests, it is also responsible for caching/uncaching data by expanding its memory footprint by allowing other remote executors utilize idle memory in node manager's own node. Secondly, Node Manager is also now responsible to provide remote node URI to executors in that node, when there is no available local memory. We discuss policies on remote node selection in Section 4.6.

d-store instances was responsible managing caching/uncaching operations by commu-

nication node manager. If Node Manager grants permission to expand its memory footprint, memory allocation and caching was performed through d-store instances. Otherwise, data was either spilled to disk, or discarded for re-computation in next iterations. With DAHI-Remote, d-store instances add another layer in their caching. In the first layer -as with DAHI-, data is written to native memory. And in the second layer, data is sent to a Remote Node Manager, whose URI is granted by Node Manager. Similarly, if there is no available memory in the cluster, d-store either spills the data to disk, or discards.

In this section, we are going to explore alternative communication protocols, and variety of transport layer implementations. We also note that our transport layer can accommodate two different environmental settings: 1) every node in the cluster is both compute and memory node, and 2) disaggregated environment in which some nodes are only used as memory node, but not compute node. In the first setting, every node participates processing Spark tasks. In the second setting, however, memory nodes are utilized in the cases when compute nodes requires more memory than they have to cache RDD partitions.

TCP/IP based solutions

TCP/IP based solutions are the most conventional and straightforward approach to develop transport layer. Spark relies on Netty [100] to establish communication between master and executors in task assignments, and communication among executors in shuffle phase. Compared to basic socket API, Netty provides non-blocking, asynchronous I/O for reading and writing data as stream. Additionally, it achieves high scalability by removing thread management in the presence of large number of connections. Nonetheless, both of the options are bounded with the limitations of TCP/IP communication. These limitations prevent us from utilizing benefits -such as low latency, high bandwidth- of emerging high speed networking technologies. Despite these limitations, TCP/IP based solutions do not require any special hardware, and can take advantage of having strong community support for the transport layer development, and further tuning and improvements.

RDMA based solutions

Remote Direct Memory Access (RDMA) is technology that allows cluster nodes exchange data without any operating system, processor and cache involvement. However, we should note that it requires special network interface card (NIC). With these features RDMA achieves faster data transfer, and existing studies shows that performance of state-of-the art high performance computing, and big data frameworks can be improved with replacing their communication layer on top of TCP/IP with RDMA. While the adoption of this technology is growing, its availability is still low compared to TCP/IP based infrastructures. Hence, in almost every framework, TCP/IP based implementation is released before its RDMA based version.

Mellanox [101] is one of the largest vendors of this technology, and in this section we are going to discuss 3 different RDMA based solutions they provide in our RDD Transport layer development.

1) Verbs API is the most primitive software interface to implement RDMA based point-to-point communication. Although this approach enables low level optimizations, to achieve that it requires deep developer understanding in not only networking, but also RDMA related concepts such as channels, connection management, message queues and access restrictions. The details of these RDMA aware network programming are outside of the scope of this section. For further details, however, user manual is accessible here [102].

2) Accelio is a high-performance, asynchronous, reliable messaging and RPC library optimized for hardware acceleration. Together with RDMA, it also supports communication over TCP/IP. It presents easy-to-use library API, and all the RDMA and TCP/IP related managements are taken care by the library transparently. It also enables having parallel connections and parallel request handling via multi-threading, resulting in high scalability. Furthermore, its asynchronous API helps us to achieve high performance, low latency communication. Finally, fault recovery is also handled by the library itself.

Figure 4.2 shows the primary layers of Accelio as follows [103] [104]:

- **Application Interface:** Provides easy-to-use primitives for fast and reliable asyn-

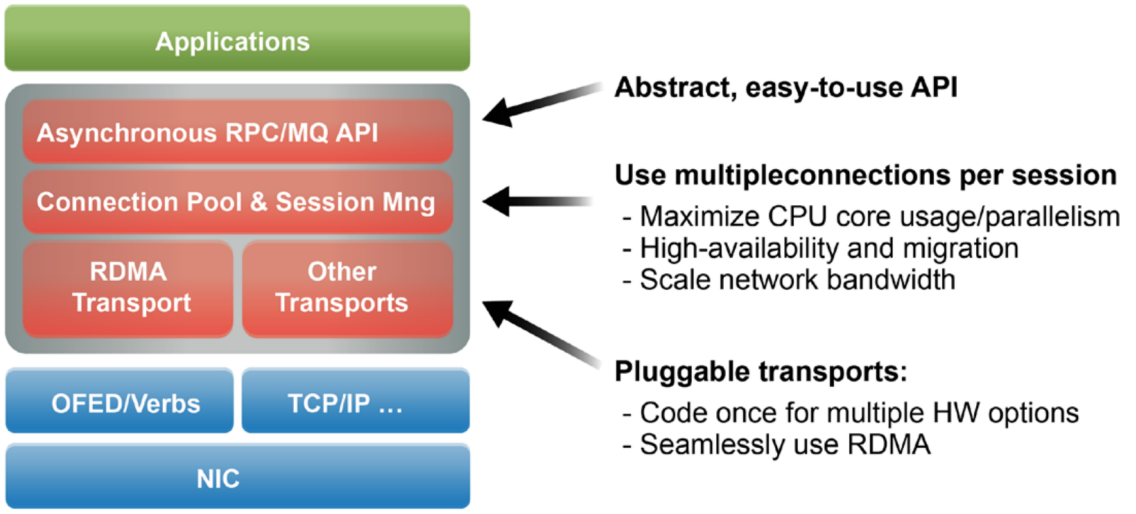


Figure 4.2: Layered Structure of Accelio Library

chronous message queue or RPC

- **Connection and Session Manager:** Delivers reliable end-to-end connectivity to peer endpoints, with dynamic connection establishment, pooling, fault recovery, and migration/redirection

- **Pluggable Transport Layer:** Enables mapping to different hardware or software transport implementations

3) NBDX, Network Block Device over Accelio is the third option we analyze for our RDD Transport Layer [39] [105].

As the name suggests, it is built on top of Accelio, as a Block Device. NBDX presents a regular storage block device to the system that can be used like any other storage (format and mount, use directly, etc.). Figure 4.3 illustrates how data transfer is maintained between client and server with NBDX. The connection and session, parallelism on top of multi-queue implementation, and channels are all handled by the library transparent to the user. However, although it is on top of Accelio, NBDX does not allow communication over TCP/IP, and since it is a block device, its API is limited by IO interface.

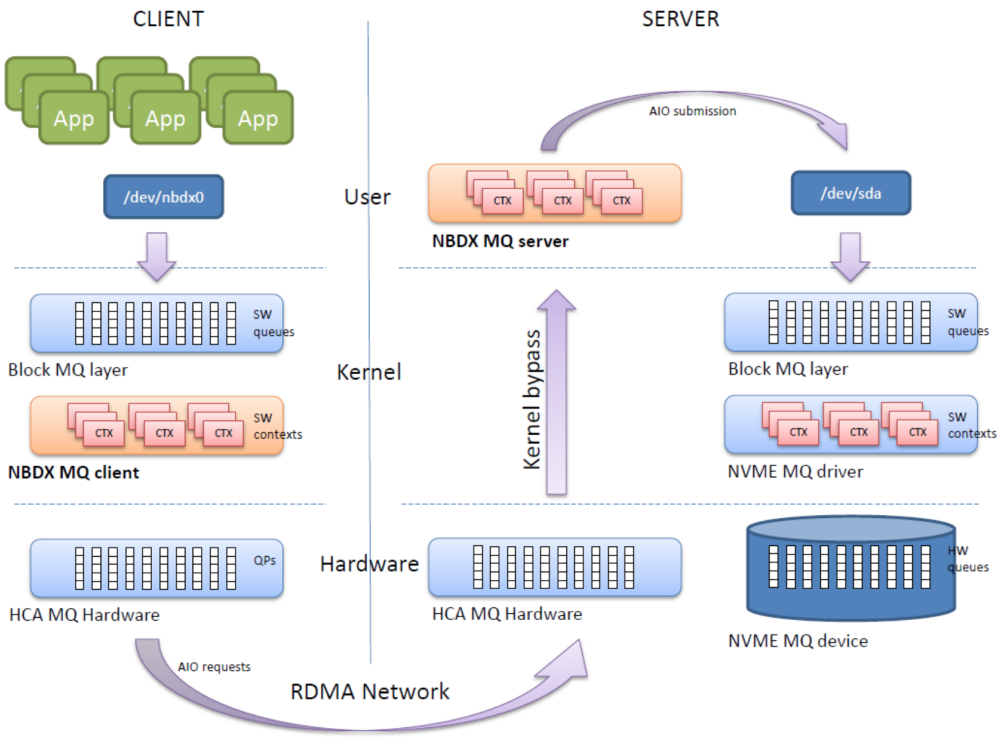


Figure 4.3: NBDX Overview

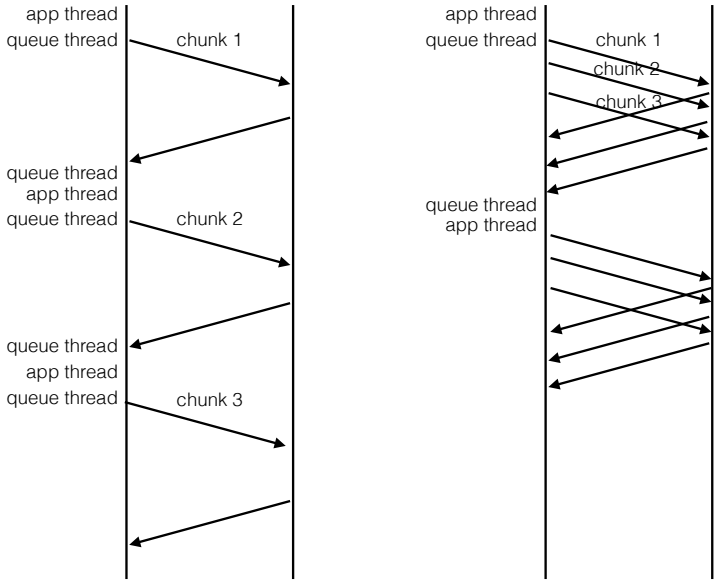
4.5 RDD Transport Layer on Top of Accelio

We select Accelio in our RDD Transport Layer development, because of its simplicity, and flexibility in configuration compared to Verbs API, and NBDX. More specifically, Accelio's capability to work with both over RDMA, and TCP/IP is the key feature we would like to have in DAHI-Remote to serve in both infrastructures.

Synchronous API

Our transport layer exposes simple KeyValue-Store-Like API to DAHI-Remote as follows:

- **put(key : string, value: byte[]) : boolean** writes the ¡string, byte array¿ pairs to remote node.
- **get(key : string) : byte[]** : reads the byte array for a given key.
- **contains(key : string) : boolean** : returns true if data exist, false otherwise.



(a) without batching (b) with batching

Figure 4.4: RDD Transport with/out Batching

- **remove(key : string) : boolean** : removes the ¡string, byte array¿ pair for a given key.

As one can notice, while Accelio is asynchronous, the calls in the API synchronous, which means application thread that executes operations for DAHI-Remote should be blocked, while Accelio queue thread is processing send/receive request/response. Using condition variables and locks, we achieved this without any busy loops. After application thread submits request(s), we put it in a sleep, and wake Accelio queue thread up. Only all the response(s) are received, we put Accelio queue thread up and wake application thread up again so it can continue to process responses.

Batching

To minimize cost of thread operations to provide synchronous API, we implemented our transport layer to support request batching of configurable number of requests. In this way, we can overlap data transfer time with request submission time. Figure 4.4 illustrates both scenarios, and in the table 4.1 we analyze the amount of time we spent on put/get requests under varying batch sizes. We also indicate the amount of space that batch queue uses for

Table 4.1: Latency of Put/Get Operations under varying Batching Sizes

Total Data Size / Queue Size	**8KB**	**64KB**	**512KB**	**4MB**	**32MB**	**256MB**
4MB	3816	3259	475	218	-	-
32MB	5402	39581	4809	1590	1449	-
256MB	44120	367552	48526	14993	12001	12687

each configuration. For example, time we spent to write 40MB of data as 8KB of chunks -which is the default max message size in Accelio- is 5402 milliseconds without batching, with batching it reduces to 1590 milliseconds, and 1449 milliseconds requiring 4MB and 32MB batch queue sizes. Considering memory consumption, and performance, we set 4MB as our batch size by default.

4.6 Core Components and Policies

Together with transport layer, DAHI-Remote also requires policies to achieve low latency, and high throughput. These policies include remote selection, stream write/read operations, partitioning and ownership updates. In this section we are going to discuss alternative techniques that can be implemented for each policy.

4.6.1 Remote node selection

With its node manager, and native memory caching, DAHI enables executors to utilize available memory in the node in coordination. Furthermore, DAHI allows executors to have fully utilize available memory by allowing them to have dynamic, on-demand memory allocation. In the cases, where node runs out of memory, we introduce DAHI-Remote to achieve cluster level memory coordination. Instead of spilling the data to disk, or discarding and recomputing in each iteration, DAHI-Remote makes use of idle memory of other nodes in the cluster.

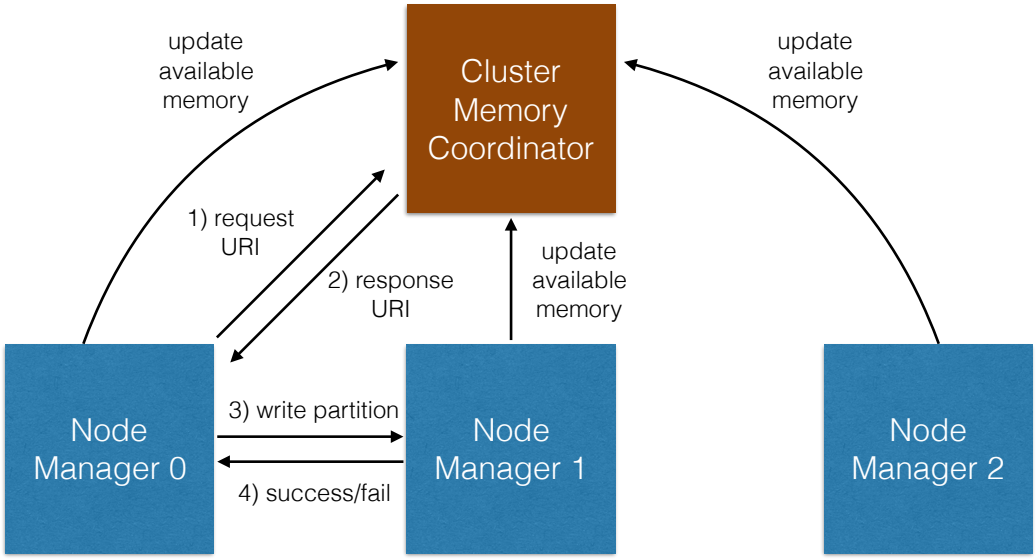


Figure 4.5: Message flow with Cluster Memory Coordinator(CMC)

Random selection is the simplest and most straightforward solution to decide which remote node to use when there is no available memory left in the current node.

Cluster Memory Coordinator (CMC) is the single point implementation for remote node selection policy. In this approach, remote node selection is a global decision, which is made by single cluster memory coordinator. When node lacks memory to cache RDD partitions, Node Manager talks to CMC which keeps track of memory utilization of each node. CMC returns Node Manager either a URI of a remote node which has available memory, or null to indicate cluster memory is fully utilized. To avoid congestion in CMC, node uses given remote node until its memory completely exhausted. Only then, node manager search for a new remote node. Similarly, when partitions are uncached, and more memory becomes available in the node, CMC is also notified and updates its ledger. In Figure 4.5 shows the flow of the information, and data among node managers, and CMC.

Hierarchical Coordination (HC) is an approach to address possible congestion problem occurs in the previous approach. Instead of every node manager talking to a single CMC, in HC node managers are grouped in hierarchical structure. In each group, there exists a group manager that keeps track of memory utilization in each node of that group.

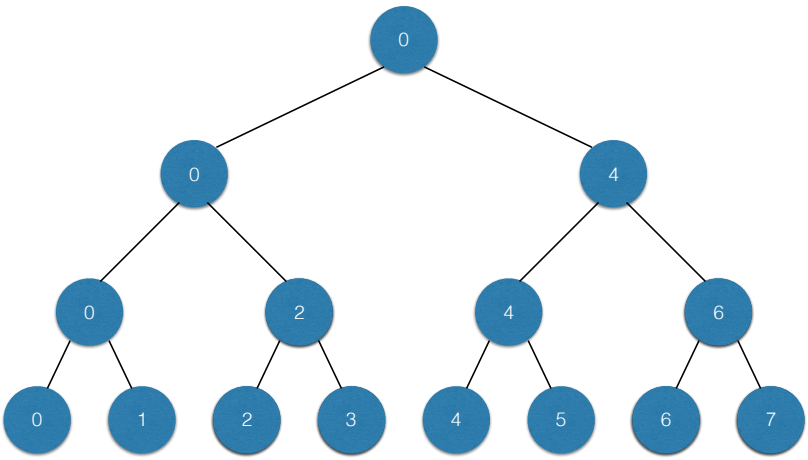


Figure 4.6: An example of Hierarchical Coordination Topology

When a node requires an additional memory, node manager talks to group manager and gets URI of a remote node with available memory. Similarly, when memory is released in a node, node manager notifies Group Manager to update amount of available memory it has. Figure 4.6 illustrates one possible Hierarchical Coordination solution.

Location Based approach assigns remote nodes based on the locations in the cluster. DAHI is already utilizes local node memory for caching. In this solution, DAHI-Remote selects one of the nodes in the rack first, and if none of the nodes in the same rack has available memory, then it returns a remote node among other cluster nodes. We should note that this approach can be bundled with one of the earlier approaches in the selection of none-rack remote node.

Ring topology can also be used for remote selection. In this approach, remote nodes are initially assigned in a way that i-th node uses ((i+1)%N)-th node as a remote node where we have N number of nodes in total in the cluster. As memory in the remote node exhausted, next node in the ring is selected as new remote node. This master-less alternative reduces the possibility of having congestion in the network. Also, minimizes the overhead of remote node selection. Figure 4.7 shows an example of ring topology.

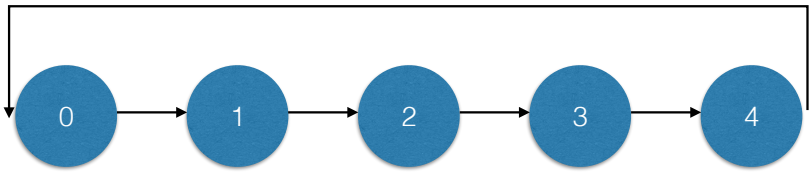


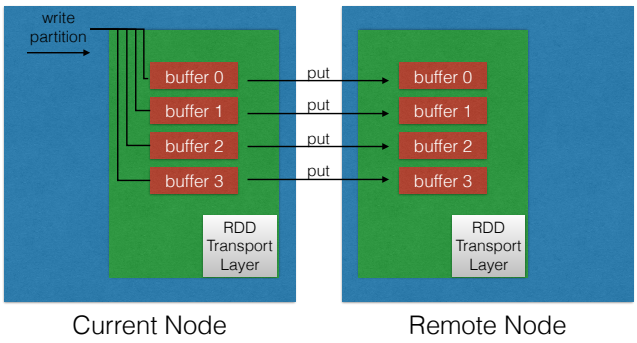
Figure 4.7: Ring Topology Based Remote Node Selection

4.6.2 Streaming Write/Read Operations

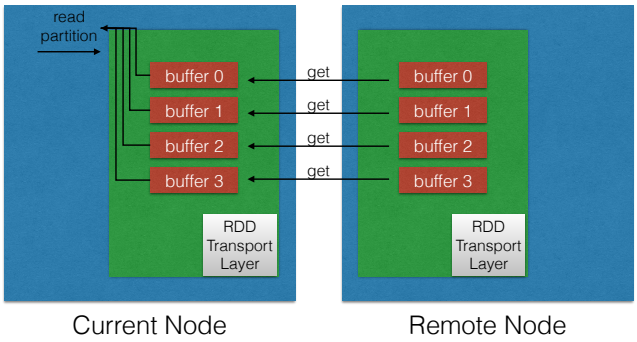
As we have stated earlier in Section 4.4, RDD Transport Layer exposes KeyValue-Store-Like API for remote put/get operations. This API requires data to be completely generated - and stored in memory- before put operation, and similarly requires data to be fetched completely with get operation so that execution can resume with the fetched data. Although it is easy-to-use, the application suffers from two main overheads as a consequence in the presence of large partitions to read/write to remote node: 1) Garbage Collection Overhead, 2) High Latency in read. To address these overheads, DAHI-Remote implements another layer of streaming on top of put/get API. Both read and write operations to remote node utilizes an internal buffer. During write, an executor fills that internal buffer as it generates RDD partition, and makes *put* call to remote for the partial data in the buffer. Similarly, during read, DAHI-Remote fetches partial data and fills the internal buffer. While an executor consumes the data in the buffer, DAHI-Remote fetches next partial data in parallel. Keeping memory utilization in smaller buffer size, instead of having an entire partition, DAHI-Remote reduces garbage collection overhead. Also, by allowing partial fetches and overlapping read and computation, it reduces the latency. In Figure 4.8, we illustrate how DAHI-Remote utilizes put/get API to achieve streaming IO operations.

4.6.3 Secondary Partitioning

Task parallelism is achieved in Spark by assigning tasks on different RDD partitions over executors. So, partitions becomes unit of computations in applications. In DAHI-Remote, we introduce secondary partitioning - for RDD partitions. Instead of writing to IO buffers



(a) Stream Write with Internal Buffers



(b) Stream Read with Internal Buffers

Figure 4.8: Streaming Write/Read Operations

created for one partition to the same remote node, DAHI-Remote distributes buffers to multiple remote nodes. Secondary partitioning aims to reduce latency caused by fetching the data from remote node. Instead of fetching data from one remote node, DAHI-Remote fetches data from multiple nodes in parallel, and feed Spark application thread. While application thread consumes the retrieved data, fetcher threads continues to bring more data from remote nodes. The level of secondary partitioning can be configured via parameter exposed in Spark configuration file.

4.6.4 Ownership Update

When there is no memory available to cache generated RDD partition in the current, DAHI-Remote ships the partition to remote node(s). In the next iterations, partition is retrieved back from the remote node, for the computation in the generated node. In this case, current node experience a delay of moving the data over the network. To address this problem,

exploiting locality aware scheduling of Spark, DAHI-Remote applies ownership update. In this approach, when partition is moved to a remote node, the ownership of the partition is changed from current node to local node, and Spark master is notified about this change. So, instead of assigning task to one node, and fetching data from another node, Spark can continue execute true locality aware task scheduling, without eliminating the overhead of data transfer among nodes.

This feature of DAHI-Remote also eliminates the need for eviction policies. Let's consider an example of three nodes, each having memory utilization at different rates. While node 0 is the most aggressive one, node 2 is the least aggressive in memory consumption. In phase one, executors in both Node 0, and Node 1 is using their local memory, but executors in Node 0 consuming memory faster than executors in Node 1. In phase 2, we see there is no available memory in Node 0, so executors ship recently generated RDD partitions to Node 1. In phase 3, there is no available memory in both of the nodes, so they both ship partitions to Node 2. In the next iteration when we read RDD partitions, with ownership update, Node 0 will work on the partitions in its local memory, similarly Node 1 and Node 2 will also work on the partitions on their local memories, regardless of which node generated those partitions.

4.6.5 Fault Tolerance

Lazy evaluation enables Spark to create directed acyclic graph (DAG) before any computation begins for a job. In this graph, vertexes represent RDDs and edges between them represents operations (map, filter, join etc.) required to compute one RDD from its parent RDDs. Fault tolerance is achieved in Spark, exploiting the advantage of this lineage graph. Since it shows how to compute missing data, from its parents, recovery can be performed without requiring any replication. DAHI-Remote also relies on the same principal. As an RDD caching mechanism, in case of failure during execution, DAHI-Remote notifies Spark so that missing but previously cached data to be recomputed.

4.7 Evaluation

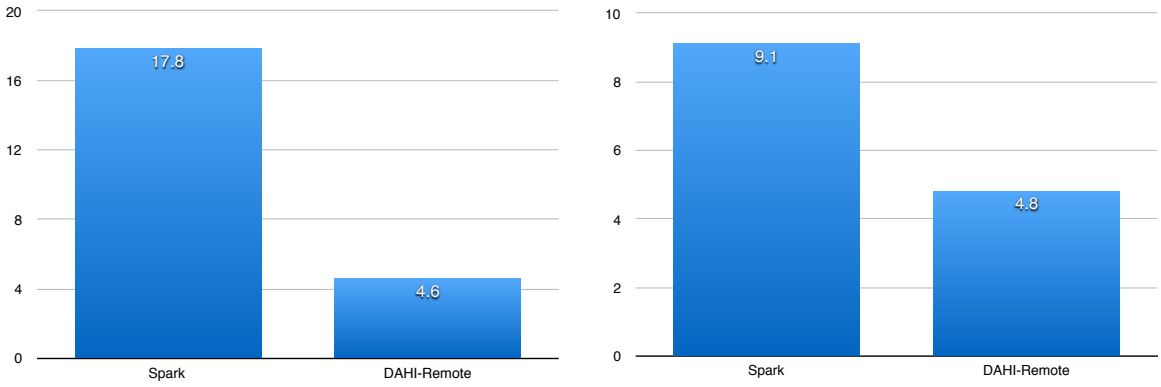
We run our experiments using public servers with RDMA network support, provided by CloudLab [106]. Each machine has 16 GB of physical memory, and 2.1 GHz Xeon E5-2450 processor, containing 8 cores. RDMA support between machines is enabled with NIC of Mellanox MX354A Dual port FDR CX3 adapter w/1 x QSA adapter.

We have evaluated both Spark, and DAHI-Remote using applications from SparkBench benchmark suite. The first application we use is linear Regression, which is a commonly used prediction algorithm that tries to determine linear relation between multi-dimensional feature vector and outcome. By evaluating the correspondence of input and output data points, it generates a model to estimate the output for any given data point. Using SparkBench, we generate the input dataset of 10 million data points, each having 100 features. The second application is connected components. For a given graph, the goal of the application is to identify vertex sets, having in each set there is a path between every pair of vertexes, and there is no path for vertexes from different sets. Using SparkBench, we generate the input graph with 1 million vertexes with parameters $\mu = 5$, $\sigma = 2$. For both of the input sets, vanilla Spark can cache 90% of RDD partition.

We configure Spark, having an 8 executors in a node, and each executor is using 1 core and 2GB of memory. In total, executors will be utilizing all the available cores, and the memory. Recall that default memory fraction for RDD caching in executors is 60%. Therefore, we configured DAHI-Remote, having an 8 executors in a node, and each executor is initialized with 0.8GB of heap. The rest of the available memory, which is 9.6GB $=$ 16GB x 60%, is coordinated by Node Manager.

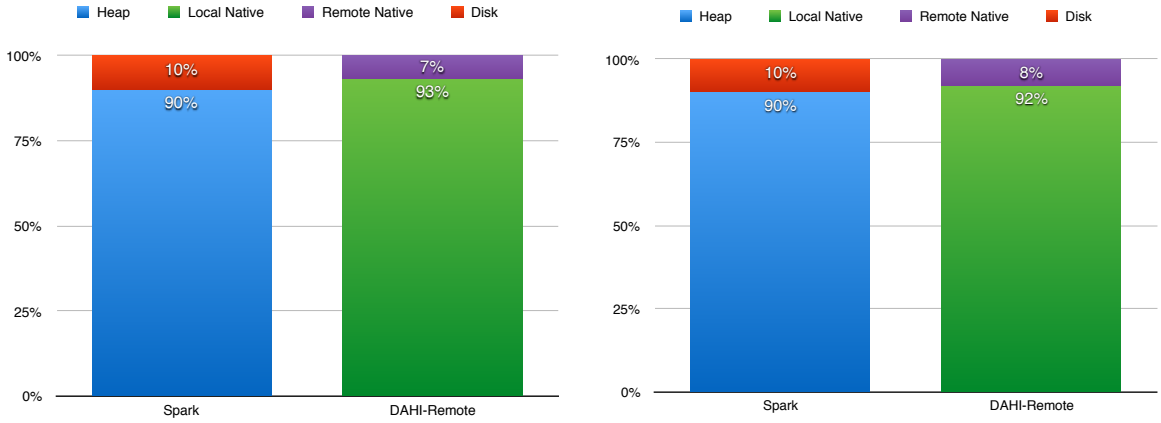
4.7.1 Spark vs DAHI-Remote

Our first set of experiments compares the performance of Vanilla Spark, with DAHI-Remote by measuring average iteration time, for both linear regression, and connected



(a) Linear Regression (b) Connected Components

Figure 4.9: Spark vs DAHI_Remote (Avg. Iteration Time in Seconds)



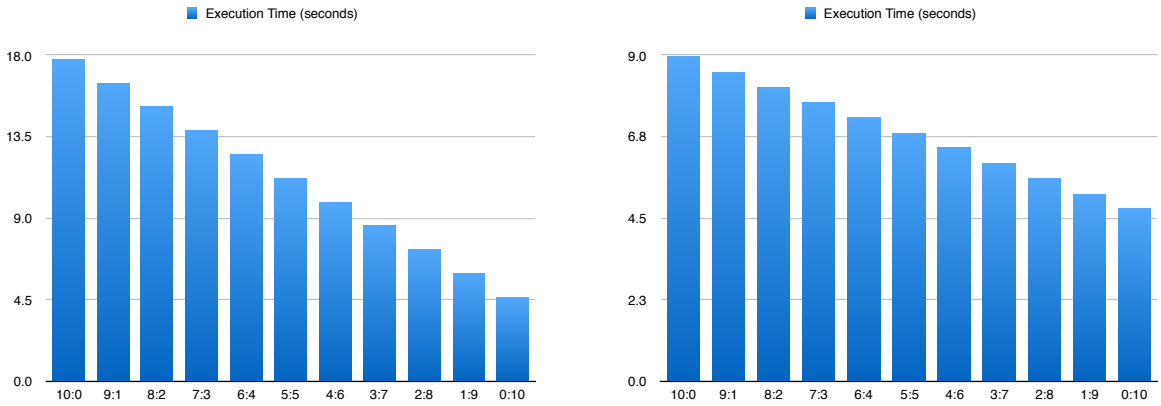
(a) Linear Regression (b) Connected Components

Figure 4.10: RDD Partition Distributions

components applications. We set number of iterations for each application to 5.

Figure 4.9 report average iterations times for LR and CC in seconds, for vanilla Spark, and DAHI-Remote. While average iteration time for LR is 17.8 seconds, DAHI-Remote is able to reduce average iteration time to 4.6 seconds. Similarly, while average iteration time for CC is 9.1 seconds, it is reduced to 4.8 seconds with DAHI-Remote. Overall speedup DAHI-Remote achieves for both applications are 3.9x, and 1.9x over vanilla Spark.

Figure 4.10 shows the distributions of the RDD partitions on heap, local native memory, remote native memory, and disk. We can say that the performance improvement has two components in DAHI-Remote. First, it uses available local memory more efficiently, so that while vanilla Spar is able to cache 90% of RDD partitions on heap, DAHI-Remote can



(a) Linear Regression (b) Connected Components

Figure 4.11: Disk/Remote Memory Ratio Experiment

Table 4.2: Average Iteration Time under Varying Disk:Remote Ratio

Disk:Remote Ratio	LR Avg. Iteration Time(seconds)	CC Avg. Iteration Time(seconds)
10:0	17.8	9.0
9:1	16.5	8.5
8:2	15.2	8.1
7:3	13.9	7.7
6:4	12.5	7.3
5:5	11.2	6.9
4:6	9.9	6.4
3:7	8.6	6.0
2:8	7.3	5.6
1:9	6.0	5.2
0:10	4.6	4.8

cache 93% and 92% of RDD partitions in local native memory for LR, and CC respectively. Second, while Spark spills the partitions -10%- that it was unable to cache in memory, DAHI-Remote utilizes remote memory for the partitions that it was unable to cache in local native memory, 7%, and 8% of partitions for LR and CC respectively.

4.7.2 DAHI-Remote Performance under Varying Remote Memory Capacity

In this set, we explore effect of remote memory capacity on the DAHI-Remote performance. Recall that when there is no available memory in the node, executors uses disk

as 3rd layer of caching. In this set, we start our experiments using remote node with no available memory -so we use disk for caching-, and gradually increase its capacity in a way that in each iteration we can increase the amount of partitions we cache in remote memory by 10%. Figure 4.11 reports our results for both LR and CC applications, showing linear performance improvement as we increase the capacity of remote memory.

4.8 Conclusion

In this chapter, we presented DAHI-Remote, as a lightweight, distributed memory coordination, and caching mechanism for JVM instances. We have evaluated the problems caused by garbage collection overhead, imbalance of memory utilization, and premature spilling as a result of the rigid structure of Spark executors. We have shown that Spark exhibits linear performance trend as input size increases linearly, up to the point that Spark can no longer achieve full partition caching. To address the potential problems of memory contention on some compute nodes in a cluster, DAHI-Remote development is aimed to alleviate the high memory pressure of those executors that cannot find sufficient idle memory on their local nodes in a cluster by creating and providing remote memory sharing opportunities from other nodes in the cluster. The DAHI-Remote framework enables memory coordination and caching among JVM instances across the entire cluster through a hierarchical RDD caching and memory sharing protocol. Caching is performed on local native memory, if available, first. Otherwise, DAHI-Remote looks for available memory in the cluster, to establish remote memory caching. Finally, if that is also not possible, partitions are either spilled to disk, or discarded for re-computation in the next iterations. We have also discussed design choices behind RDD transport layer that DAHI-Remote implements, including available policies for remote node selection, and fault tolerance. DAHI-Remote also aims to reduce data movement across nodes, by implementing ownership update. With that, task on the data that has been cached in remote node, is no longer assigned on the executors in the node that generates it. Instead, ownership of the partition is changed to the node that caches it.

This way, DAHI-Remote can work well with existing locality aware scheduling of Spark. Finally, we have conducted our experiments using Linear Regression and Connected Components, and we have shown that performance of these applications have been improved by DAHI-Remote by 3.9x and 1.9x compared to Vanilla Spark. We have also shown the impact of remote memory capacity, under varying disk:remote memory ratios.

CHAPTER 5
CONCLUSION

Most of the big data frameworks are built on top of Java Virtual Machines. Together with it's great community support, and developer friendliness, one of the main reasons behind this choice is JVM's removing burden of explicit memory management, such as handling memory leaks, dangling pointers, via its garbage collection mechanism. However, despite of its effectiveness in framework development, executors may suffer from high garbage collection overhead, especially running memory greedy applications, or there exists a memory pressure in the system. Or even worse, system may crash throwing out-of-memory error. This dissertation uses Spark, as an example of JVM based large scale data processing framework, and addresses issues problems caused by JVM itself, or the way framework uses them as executors. For example, Spark launches JVM executors with a fixed size of memory. In the presence of unbalanced memory utilization however, an executor which requires more memory to cache data cannot utilize idle memory of another executor in the same node. Hence, depending on the predefined decision, data is either spilled to disk, or discarded for re-computation every time it is needed. Moreover, the unbalanced memory utilization can occur in cluster level, when executors in node may require more memory, but cannot utilize available memory in another node.

First, we explored how JVM manages its heap -in terms of generations-, and available garbage collection alternatives. Then we studied how parameter configuration affects garbage collection overhead, and total execution time using different memory intensive workloads from different benchmark suites. Our experimental study has shown that increasing heap size improves application performance up to a certain point, by reducing garbage collection overhead. It has also shown that applications can throw out-of-memory exception, while they have large portion of unused memory reserved survivor objects. Fi-

nally, we have shown the same application performance can be achieved with smaller heap sizes, by only configuring heap structure and garbage collector.

Second, we analyzed the use of JVMs in big data frameworks, using Spark as an example. Our experiments have shown the benefits of RDD caching to improve iterative applications when memory is plenty, also reported opportunities to make memory management more efficient for the scenarios where RDD can only partially cached in the available memory. Launching JVM executors with fixed amount of memory prevents them to have dynamic, and elastic on-demand memory allocation. Hence, an executor that requires more memory is not be able to utilize idle memory of another executor. This causes Spark to spill overflowed partitions to either spill to disk, or discard for re-computation, hurting overall application performance significantly. We proposed DAHI, as a lightweight, memory coordination and caching mechanism to address these problems. We compared the performance of DAHI, and Vanilla Spark by using benchmarks from machine learning, and graph processing domains using workloads from SparkBench. We have shown that performance of JVM executors can be improved when they are able to have dynamic on-demand memory utilization. This elasticity is granted by DAHI with caching on native memory and coordination with other executors in the system. We have shown that while caching on native memory reduces garbage collection overhead, coordination with other executors allows executors to utilize total available memory to achieve higher data caching, resulting in overall performance improvement in iterative Spark applications.

Third, we proposed DAHI-Remote by extending DAHI, with the capability of cluster level memory coordination and caching. With DAHI-Remote, we now have layered structure for RDD caching, in which first local native memory memory is used with original DAHI, and idle remote memory as second layer. And if there is no available memory in the entire cluster, partition is either spilled to disk, or discarded. We also discussed several design choices for RDD transport layer, including its implementation on top of Accelio library, remote node selection alternatives, streaming write/read operations to reduce la-

tency, and ownership updates to minimize data movement among nodes after caching. We compared the performance of DAHI-Remote with Spark in a disaggregated cluster, where a subset of nodes are used as compute nodes, and others as memory nodes. We have shown that performance of JVM executors can be improved by utilizing available remote memory for overflowed partitions, instead of spilling them to disk. We have also shown that as we increase the amount of available remote memory, average iteration time decreases linearly, until we cache every partition in both local and remote memory.

Appendices

APPENDIX A

EXPERIMENTAL EQUIPMENT

The experiments in Chapter 2 are conducted in following setup:

- **Java Version:** Oracles HotSpot Java Runtime Environment version 1.8.0_65
- **Operating System:** Mac OSX Yosemite 10.10.5
- **Processor:** 1 x Intel Core i5 (2 cores, 2.9 GHz)
- **Memory:** 8 GB 1867 MHz DDR3

The experiments in Chapter 3 are conducted in following setup:

- **Host:** zhuhai.cc.gatech.edu
- **Spark Version:** 1.6.1
- **Java Version:** OpenJDK Runtime Environment version 1.8.0_141
- **Processor:** 1 x Intel Xeon (8 cores, 2.67 GHz)
- **Memory:** 96 GB

The experiments in Chapter 4 are conducted in following setup:

- **Host:** r320 Machines in CloudLab [106]
- **Spark Version:** 1.6.1
- **Java Version:** OpenJDK Runtime Environment version 1.8.0_141
- **Processor:** 1 x Intel Xeon E5-2450 (8cores, 2.1 GHz)
- **Memory:** 16GB Memory (4 x 2GB RDIMMs, 1.6Ghz)

- **NIC:** 1GbE Dual port embedded NIC (Broadcom)
- **NIC:** 1 x Mellanox MX354A Dual port FDR CX3 adapter w/1 x QSA adapter

APPENDIX B

BENCHMARKS

The benchmarks used in Chapter 2 are as follows:

- h2 from DaCapo benchmark suite
- derby, serial, and compiler.compiler from SPECjvm2008 benchmark suite

The benchmarks used in Chapter 3 are as follows:

- Linear Regression from SparkBench
- SVM from SparkBench
- K-Means from SparkBench
- Connected Components from SparkBench

The benchmarks used in Chapter 4 are as follows:

- Linear Regression from SparkBench
- Connected Components from SparkBench

REFERENCES

[1] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, pp. 74–80, 2013.

[2] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer*, pp. 103–105, 2007.

[3] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, 2012, 9:1–9:14.

[4] Y. Xu, M. Bailey, B. Noble, and F. Jahanian, "Small is better: Avoiding latency traps in virtualized data centers," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, 2013, 7:1–7:16.

[5] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda, "Less is more: Trading a little bandwidth for ultra-low latency in the data center," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, 2012, pp. 19–19.

[6] K. Nichols and V. Jacobson, "Controlling queue delay," *Queue*, 20:20–20:34, 2012.

[7] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11, 2011, pp. 50–61.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, pp. 205–220, 2007.

[9] R. C. Chiang and H. H. Huang, "Tracon: Interference-aware scheduling for data-intensive applications in virtualized environments," in *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[10] N. Struckmann, Y. Sandoval, N. Har'El, F. Chen, S. Fan, J. Činkelj, G. Berginc, P. Chronz, N. Gilboa, G. Scalosub, K. Meth, and J. Kennedy, "Mikelangelo: Micro kernel virtualization for high performance cloud and hpc systems," in *Advances in Service-Oriented and Cloud Computing*, Z. Á. Mann and V. Stolz, Eds., Springer International Publishing, 2018, pp. 175–180.

[11] P. Bodík, I. Menache, M. Chowdhury, P. Mani, D. A. Maltz, and I. Stoica, "Surviving failures in bandwidth-constrained datacenters," in *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '12, 2012, pp. 431–442.

[12] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," in *In USENIX OSDI*, 2008.

[13] C. A. Reiss, "Understanding memory configurations for in-memory analytics.," PhD thesis, University of California, Berkeley, 2016.

[14] M. Hines, A. Gordon, M. Silva, D. da Silva, K. D. Ryu, and M. Ben-Yehuda, "Applications know best: Performance-driven memory overcommit with ginkgo," in *CloudCom*, 2011.

[15] T. Wood, G. Tarasuk-Levin, P. Shenoy, P. Desnoyers, E. Cecchet, and M. D. Corner, "Memory buddies: Exploiting page sharing for smart colocation in virtualized data centers," in *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '09, 2009, pp. 31–40.

[16] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13, 2013, 5:1–5:16.

[17] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12, 2012, 7:1–7:13.

[18] H. Garcia-Molina and K. Salem, "Main memory database systems: An overview," *IEEE Transactions on Knowledge and Data Engineering*, no. 6, pp. 509–516, 1992.

[19] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, 1984, pp. 1–8.

[20] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, 2004.

[21] F. M. Cuenca-Acuna and T. D. Nguyen, "Cooperative caching middleware for cluster-based servers," in *Proceedings 10th IEEE International Symposium on High Performance Distributed Computing*, 2001, pp. 303–314.

[22] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 385–398.

[23] J. Gray and F. Putzolu, "The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time," *SIGMOD Rec.*, pp. 395–398, 1987.

[24] R. Murphy, A. Rodrigues, P. Kogge, and K. Underwood, "The implications of working set analysis on supercomputing memory hierarchy design," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05, 2005, pp. 332–340.

[25] I. Zhang, A. Garthwaite, Y. Baskakov, and K. C. Barr, "Fast restore of checkpointed memory using working set estimation," in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '11, 2011, pp. 87–98.

[26] P. J. Denning, "The working set model for program behavior," *Commun. ACM*, pp. 323–333, 1968.

[27] E. Albert, S. Genaim, and M. Gómez-Zamalloa Gil, "Live heap space analysis for languages with garbage collection," in *Proceedings of the 2009 International Symposium on Memory Management*, ser. ISMM '09, 2009, pp. 129–138.

[28] Intel, *Intel data plane development kit: Getting started guide*.

[29] *Open vswitch*, `http://www.openvswitch.org`, accessed in 2013.

[30] *Vmware white paper. tributed switch*, 2013.

[31] *Hp: The machine*, `http://www.labs.hpe.com/research/themachine`.

[32] *Intel rsa*, `http://www.intel.com/content/www/us/en/architecture-and-technology/rsa-demo-x264.html`.

[33] K. T. P. Lim, J. Chang, T. N. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch, "Disaggregated memory for expansion and sharing in blade servers," in *ISCA*, 2009.

[34] K. Lim, Y. Turner, J. R. Santos, A. AuYoung, J. Chang, P. Ranganathan, and T. F. Wenisch, "System-level implications of disaggregated memory," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012, pp. 1–12.

[35] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 249–264.

[36] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, P. Subrahmanyam, L. Suresh, K. Tati, R. Venkatasubramanian, and M. Wei, "Remote memory in the age of fast networks," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17, 2017, pp. 121–127.

[37] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "Latency-tolerant software distributed shared memory," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 291–305.

[38] T. Newhall, S. Finney, K. Ganchev, and M. Spiegel, "Nswap: A network swapping module for linux clusters," in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., 2003, pp. 1160–1169.

[39] "Accelio based network block device," Retrieved March 2019 from `https://community.mel` 2019.

[40] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[41] J. S. Vitter, "External memory algorithms and data structures: Dealing with massive data," *ACM Comput. Surv.*, pp. 209–271, 2001.

[42] C. Inc., *Cray xt4 and xt3 datasheet*.

[43] W. Zhao, Z. Wang, and Y. Luo, "Dynamic memory balancing for virtual machines," *SIGOPS Oper. Syst. Rev.*, 2009.

[44] Q. Zhang, L. Liu, C. Pu, W. Cao, and S. Sahin, "Efficient shared memory orchestration towards demand driven memory slicing," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1212–1223.

[45] W. Cao, L. Liu, C. Pu, S. Sahin, and Y. Wu, "Efficient host and remote memory sharing with fastswap," in *Technical Report*, 2018.

[46] Y. Ren, L. Liu, Q. Zhang, Q. Wu, J. Guan, J. Kong, H. Dai, and L. Shao, "Shared-memory optimizations for inter-virtual-machine communication," *ACM Comput. Surv.*, 49:1–49:42, 2016.

[47] Q. Zhang, L. Liu, G. Su, and A. Iyengar, "Memflex: A shared memory swapper for high performance vm execution," *IEEE Transactions on Computers*, pp. 1645–1652, 2017.

[48] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," *CoRR*, 2018.

[49] H. Kim, H. Jo, and J. Lee, "Xhive: Efficient cooperative caching for virtual machines," *IEEE Transactions on Computers*, pp. 106–119, 2011.

[50] J. Higgins, V. Holmes, and C. Venters, "Orchestrating docker containers in the hpc environment," in *High Performance Computing*, J. M. Kunkel and T. Ludwig, Eds., 2015, pp. 506–513.

[51] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," ser. OSDI, 2006.

[52] M. Hertz, S. Kane, E. Keudel, T. Bai, C. Ding, X. Gu, and J. E. Bard, "Waste not, want not: Resource-based garbage collection in a shared environment," *SIGPLAN Not.*, 2011.

[53] N. Bobroff, P. Westerink, and L. Fong, "Active control of memory for java virtual machines and applications," in *ICAC*, 2014.

[54] M. Hertz and E. D. Berger, "Quantifying the performance of garbage collection vs. explicit memory management," *SIGPLAN Not.*, pp. 313–326, 2005.

[55] "Apache spark project," `http://spark.apache.org`.

[56] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 15–28.

[57] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European Conference on Computer Systems*, ser. EuroSys '10, 2010, pp. 265–278.

[58] S. Sahin, W. Cao, Q. Zhang, and L. Liu, "Jvm configuration management and its performance impact for big data applications," in *2016 IEEE International Congress on Big Data (BigData Congress)*, 2016, pp. 410–417.

[59] "Apache hadoop map reduce," `http://hadoop.apache.org`.

[60] Storm, "Apache Storm project," Retrieved March 2016 from http://storm-project.net/ 2016.

[61] M. Hertz, Y. Feng, and E. D. Berger, "Garbage collection without paging," in *ACM SIGPLAN PLDI*, 2005.

[62] C. A. Waldspurger, "Memory resource management in vmware esx server," *SIGOPS Oper. Syst. Rev.*, 2002.

[63] T.-I. Salomie, G. Alonso, T. Roscoe, and K. Elphinstone, "Application level ballooning for efficient server consolidation," ser. EuroSys, 2013.

[64] R. Mcdougall, W. Huang, and B. Corrie, *Cooperative memory resource management via application-level balloon*, US Patent App. 12/826,389, 2011.

[65] N. Mitchell and G. Sevitsky, "The causes of bloat, the limits of health," ser. OOPSLA, 2007.

[66] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," *SIGPLAN Not.*, 2013.

[67] "Visual vm," https://visualvm.java.net/.

[68] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *ACM SIGPLAN OOPSLA*, 2006.

[69] Standard Performance Evaluation Corporation, "Specjvm2008," https://www.spec.org/jv 2016.

[70] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM (CACM)*, vol. 51, no. 1, pp. 107–113, 2008.

[71] MPI, "Mpi - message passing interface," http://mpi-forum.org/docs/mpi-3.1/mpi3 2016.

[72] L. Lu, X. Shi, Y. Zhou, X. Zhang, H. Jin, C. Pei, L. He, and Y. Geng, "Lifetime-based memory management for distributed data processing systems," *Proc. VLDB Endow.*, 2016.

[73] Redis, "Redis," https://redis.io/, 2018.

[74] S. Alsubaiee, Y. Altowim, H. Altwaijry, A. Behm, V. Borkar, Y. Bu, M. Carey, I. Cetindil, M. Cheelangi, K. Faraaz, E. Gabrielova, R. Grover, Z. Heilbron, Y.-S. Kim, C. Li, G. Li, J. M. Ok, N. Onose, P. Pirzadeh, V. Tsotras, R. Vernica, J. Wen, and T. Westmann, "Asterixdb: A scalable, open source bdms," *Proc. VLDB Endow.*, 2014.

[75] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, "Tachyon: Reliable, memory speed storage for cluster computing frameworks," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14, 2014.

[76] X. Lu, D. Shankar, S. Gugnani, and D. K. D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016.

[77] B. Nicolae, C. Costa, C. Misale, K. Katrinis, and Y. Park, "Towards memory-optimized data shuffling patterns for big data analytics," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[78] B. Nicolae, C. H. A. Costa, C. Misale, K. Katrinis, and Y. Park, "Leveraging adaptive i/o to optimize collective data shuffling patterns for big data analytics," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[79] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11, 2011, pp. 295–308.

[80] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle, "The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing," *Proc. VLDB Endow.*, pp. 1792–1803, 2015.

[81] S. Şahin and B. Gedik, "C-stream: A co-routine-based elastic stream processing engine," *ACM Trans. Parallel Comput.*, 15:1–15:27, 2018.

[82] A. M. Aly, A. Sallam, B. M. Gnanasekaran, L. Nguyen-Dinh, W. G. Aref, M. Ouzzani, and A. Ghafoor, "M3: Stream processing on main-memory mapreduce," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 1253–1256.

[83] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, 2013, pp. 423–438.

[84] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, pp. 1626–1629, 2009.

[85] *Hbase*, `https://hbase.apache.org/`, accessed in 2018.

[86] A. Richard, L. Nguyen, P. Shipton, K. B. Kent, A. Bierbrauer, K. Nasartschuk, and M. Dombrowski, "Inter-jvm sharing," *Software: Practice and Experience*, pp. 1285–1296, 2016.

[87] J. Simão and L. Veiga, "Qoe-jvm: An adaptive and resource-aware java runtime for cloud computing," in *On the Move to Meaningful Internet Systems: OTM 2012*, R. Meersman, H. Panetto, T. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. F. Cruz, Eds., 2012, pp. 566–583.

[88] D. Chen, C. Tang, S. Dwarkadas, and M. L. Scott, "Jvm for a heterogeneous shared memory system," in *IN PROC. OF THE WORKSHOP ON CACHING, COHERENCE, AND CONSISTENCY (WC3 02*, 2002.

[89] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster, "A high performance cluster jvm presenting a pure single system image," in *Proceedings of the ACM 2000 Conference on Java Grande*, 2000, pp. 168–177.

[90] M. Lobosco, A. Silva, O. Loques, and C. L. de Amorim, "A new distributed jvm for cluster computing," in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., Springer Berlin Heidelberg, 2003, pp. 1207–1215.

[91] and and F. C. M. Lau, "Efficient global object space support for distributed jvm on cluster," in *Proceedings International Conference on Parallel Processing*, 2002, pp. 371–378.

[92] X. Lu, N. S. Islam, M. Wasi-Ur-Rahman, J. Jose, H. Subramoni, H. Wang, and D. K. Panda, "High-performance design of hadoop rpc with rdma over infiniband," in *2013 42nd International Conference on Parallel Processing*, 2013, pp. 641–650.

[93] M. Wasi-ur-Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. D. Panda, "High-performance rdma-based design of hadoop mapreduce over infiniband," in *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, 2013, pp. 1908–1917.

[94] X. Lu, D. Shankar, S. Gugnani, and D. K. D. K. Panda, "High-performance design of apache spark with rdma and its benefits on various workloads," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 253–262.

[95] N. S. Islam, D. Shankar, X. Lu, M. Wasi-Ur-Rahman, and D. K. Panda, "Accelerating i/o performance of big data analytics on hpc clusters through rdma-based key-value store," in *2015 44th International Conference on Parallel Processing*, 2015, pp. 280–289.

[96] *Apache ignite*, `https://ignite.apache.org/`, accessed in 2018.

[97] *Hazelcast*, `https://hazelcast.com/`, accessed in 2018.

[98] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The case for ramclouds: Scalable high-performance storage entirely in dram," *SIGOPS Oper. Syst. Rev.*, pp. 92–105, 2010.

[99] S. Zhang, J. Han, Z. Liu, K. Wang, and S. Feng, "Accelerating mapreduce with distributed memory cache," in *2009 15th International Conference on Parallel and Distributed Systems*, 2009, pp. 472–478.

[100] "Netty," Retrieved March 2019 from `https://netty.io/`, 2019.

[101] "Meallanox technologies," Retrieved March 2019 from `http://www.mellanox.com/`, 2019.

[102] "Rdma aware networks programming user manual," Retrieved March 2019 from `http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Pro` 2019.

[103] "Accelio," Retrieved March 2019 from `http://www.accelio.org`, 2019.

[104] "Accelio source code," Retrieved March 2019 from `https://github.com/accelio/acce` 2019.

[105] "Nbdx source code," Retrieved March 2019 from `https://github.com/accelio/NBDX`, 2019.

[106] *Cloudlab*, `https://www.cloudlab.us/`, accessed in 2018.

VITA

Semih Sahin was born in a beautiful city of Eskisehir, in Turkey. He moved to Istanbul for his high school education to a boarding school with a full scholarship, where he met programming, algorithms and data structures. He won bronze medal in National Olympiads of Informatics in his second year. Then he moved to Ankara, the capital city of Turkey, to attend Bilkent University, and received his Bachelor of Science, and Master of Science degree there. Subsequently, he moved to Atlanta to pursue a Ph.D. in Computer Science at the College of Computing at Georgia Institute of Technology. As a member of the DiSL research group, he conducted research on various aspects of big data systems, and machine learning frameworks, under the guidance of Professor Ling Liu. His research has resulted in numerous publications that have appeared in various international conferences and journals.